

大数据

宋智军 编著

清华大学出版社

深入浅出系列规划教材

深入浅出大数据

宋智军　编著

清 华 大 学 出 版 社
北　　京

内 容 简 介

本书坚持以大数据基础和应用为主导的编写原则，理论联系实际，并通过大量实例循序渐进地为读者介绍了进行大数据实践所涉及的各类知识。为了更好地帮助读者在短时间内掌握大数据基础理论知识和实践能力，全书的基础知识介绍清晰，理论联系实际，具有很强的操作性，并提供了大量通过测试可运行的完整实例，这些实例都给出了设计步骤、代码详解及程序运行结果，对于容易出现问题的地方，则以“注”的方式介绍常用的技巧和注意事项。另外本书的配套资料可从清华大学出版社网站(www.tup.com.cn)上下载。

本书可作为计算机专业的本科生和研究生的大数据基础教材，也可作为大数据技术培训、Hadoop应用开发和运行维护人员的必备参考书。

图书在版编目（CIP）数据

深入浅出大数据/宋智军编著.--北京：清华大学出版社，2016
深入浅出系列规划教材
ISBN 978-7-302-42181-8

Ⅰ.①深…　Ⅱ.①宋…　Ⅲ.①数据处理－教材　Ⅳ.①TP274

中国版本图书馆 CIP 数据核字(2015)第 272725 号

责任编辑：白立军　薛　阳
封面设计：傅瑞学
责任校对：焦丽丽
责任印制：刘海龙

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：三河市君旺印务有限公司
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：24　　**字　　数**：553 千字
版　　次：2016 年 3 月第 1 版　　**印　　次**：2016 年 3 月第1 次印刷
印　　数：1～2000
定　　价：49.00 元

产品编号：062976-01

丛书序

为什么开发深入浅出系列丛书？

目的是从读者角度写书，开发出高质量的、适合阅读的图书。

“不积跬步，无以至千里；不积小流，无以成江海。”知识的学习是一个逐渐积累的过程，只有坚持系统地学习知识，深入浅出，坚持不懈，持之以恒，才能把一类技术学习好。坚持的动力源于所学内容的趣味性和讲法的新颖性。

计算机课程的学习也有一条隐含的主线，那就是“提出问题→分析问题→建立数学模型→建立计算模型→通过各种平台和工具得到最终正确的结果”，培养计算机专业学生的核心能力是“面向问题求解的能力”。由于目前大学计算机本科生培养计划的特点，以及受教学计划和课程设置的原因，计算机科学与技术专业的本科生很难精通掌握一门程序设计语言或者相关课程。各门课程设置比较孤立，培养的学生综合运用各方面的知识能力方面有欠缺。传统的教学模式以传授知识为主要目的，能力培养没有得到充分的重视。很多教材受教学模式的影响，在编写过程中，偏重概念讲解比较多，而忽略了能力培养。为了突出内容的案例性、解惑性、可读性、自学性，本套书努力在以下方面做好工作。

1. 案例性

所举案例突出与本课程的关系，并且能恰当反映当前知识点。例如，在计算机专业中，很多高校都开设了高等数学、线性代数、概率论，不言而喻，这些课程对于计算机专业的学生来说是非常重要的，但就目前对不少高校而言，这些课程都是由数学系的老师讲授，教材也是由数学系的老师编写，由于学科背景不同和看待问题的角度不同，在这些教材中基本都是纯数学方面的案例，作为计算机系的学生来说，学习这样的教材缺少源动力并且比较乏味，究其原因，很多学生不清楚这些课程与计算机专业的关系是什么。基于此，在编写这方面的教材时，可以把计算机上的案例加入其中，例如，可以把计算机图形学中的三维空间物体图像在屏幕上的伸缩变换、平移变换和旋转变换在矩阵运算中进行举例；可以把双机热备份的案例融入到马尔科夫链的讲解；把密码学的案例融入到大数分解中等。

2. 解惑性

很多教材中的知识讲解注重定义的介绍，而忽略因果性、解释性介绍，往往造成知其然而不知其所以然。下面列举两个例子。

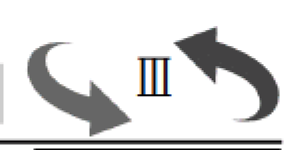
(1) 读者可能对 OSI 参考模型与 TCP/IP 参考模型的概念产生混淆，因为两种模型之

间有很多相似之处。其实,OSI 参考模型是在其协议开发之前设计出来的,也就是说,它不是针对某个协议族设计的,因而更具有通用性。而 TCP/IP 模型是在 TCP/IP 协议栈出现后出现的,也就是说,TCP/IP 模型是针对 TCP/IP 协议栈的,并且与 TCP/IP 协议栈非常吻合。但是必须注意,TCP/IP 模型描述其他协议栈并不合适,因为它具有很强的针对性。说到这里读者可能更迷惑了,既然 OSI 参考模型没有在数据通信中占有主导地位,那为什么还花费这么大的篇幅来描述它呢?其实,虽然 OSI 参考模型在协议实现方面存在很多不足,但是,OSI 参考模型在计算机网络的发展过程中起到了非常重要的作用,并且,它对未来计算机网络的标准化、规范化的发展有很重要的指导意义。

(2) 再例如,在介绍原码、反码和补码时,往往只给出其定义和举例表示,而对最后为什么在计算机中采取补码表示数值?浮点数在计算机中是如何表示的?字节类型、短整型、整型、长整型、浮点数的范围是如何确定的?下面我们来回答这些问题(以 8 位数为例),原码不能直接运算,并且 0 的原码有+0 和-0 两种形式,即 00000000 和 10000000,这样肯定是不行的,如果根据原码计算设计相应的门电路,由于要判断符号位,设计的复杂度会大大增加,不合算;为了解决原码不能直接运算的缺点,人们提出了反码的概念,但是 0 的反码还是有+0 和-0 两种形式,即 00000000 和 11111111,这样是不行的,因为计算机在计算过程中,不能判断遇到 0 是+0 还是-0;而补码解决了 0 表示的唯一性问题,即不会存在+0 和-0,因为+0 是 00000000,它的补码是 00000000,-0 是 10000000,它的反码是 11111111,再加 1 就得到其补码是 100000000,舍去溢出量就是 00000000。知道了计算机中数用补码表示和 0 的唯一性问题后,就可以确定数据类型表示的取值范围了,仍以字节类型为例,一个字节共 8 位,有 00000000~11111111 共 256 种结果,由于 1 位表示符号位,7 位表示数据位,正数的补码好说,其范围从 00000000~011111111,即 0~127;负数的补码为 10000000~11111111,其中,11111111 为-1 的补码,10000001 为-127 的补码,那么到底 10000000 表示什么最合适呢?8 位二进制数中,最小数的补码形式为 10000000;它的数值绝对值应该是各位取反再加 1,即为 01111111+1=10000000=128,又因为是负数,所以是-128,即其取值范围是-128~127。

3. 可读性

图书的内容要深入浅出,使人爱看、易懂。一本书要做到可读性好,必须做到“善用比喻,实例为王”。什么是深入浅出?就是把复杂的事物简单地描述明白。把简单事情复杂化的是哲学家,而把复杂的问题简单化的是科学家。编写教材时要以科学家的眼光去编写,把难懂的定义,要通过图形或者举例进行解释,这样能达到事半功倍的效果。例如,在数据库中,第一范式、第二范式、第三范式、BC 范式的概念非常抽象,很难理解,但是,如果以一个教务系统中的学生表、课程表、教师表之间的关系为例进行讲解,从而引出范式的概念,学生会比较容易接受。再例如,在生物学中,如果纯粹地讲解各个器官的功能会比较乏味,但是如果提出一个问题,如人的体温为什么是 37℃?以此为引子引出各个器官的功能效果要好得多。再例如,在讲解数据结构课程时,由于定义多,表示抽象,这样达不到很好的教学效果,可以考虑在讲解数据结构及其操作时用程序给予实现,让学生看到直接的操作结果,如压栈和出栈操作,可以把 PUSH()和 POP()操作实现,这样效果会好



很多，并且会激发学生的学习兴趣。

4. 自学性

一本书如果适合自学学习，对其语言要求比较高。写作风格不能枯燥无味，让人看一眼就拒人千里之外，而应该是风趣、幽默，重要知识点多举实际应用的案例，说明它们在实际生活中的应用，应该有画龙点睛的说明和知识背景介绍，对其应用需要注意哪些问题等都要有提示等。

一书在手，从第一页开始的起点到最后一页的终点，如何使读者能快乐地阅读下去并获得知识？这是非常重要的问题。在数学上，两点之间的最短距离是直线。但在知识的传播中，使读者感到"阻力最小"的书才是好书。如同自然界中没有直流的河流一样，河水在重力的作用下一定沿着阻力最小的路径向前进。知识的传播与此相同，最有效的传播方式是传播起来损耗最小，阅读起来没有阻力。

是为序。

欢迎老师投稿：bailj@tup.tsinghua.edu.cn。

2014 年 12 月 15 日

前　言

随着互联网的快速发展以及云计算、物联网和移动互联网等新一代信息技术的广泛应用，全球数据总量规模呈指数级增长。根据麦肯锡全球研究院（MGI）预测：目前，全球数据总量规模已达到ZB（泽字节）级别；预计到2020年，人类所产生的数据量将超过40ZB（4×10^{13}GB）。因此，新一轮的信息消费热潮已爆发，大数据时代已经悄然来到人们身边。

大数据无疑是当前社会发展的热点，如何借助大数据技术来改变企业信息化建设的现状，如何利用大数据进行商业模式、业务模式及经营模式的创新和变革，是当下迫切需要解决的问题。这就要求我们掌握好大数据技术，为企业信息化建设添砖加瓦。由于大数据技术对个人的综合能力要求较高，而对于初学者来说还不知道如何进入大数据殿堂。本书为读者打开了大数据领域的大门，帮助初学者建立大数据的基本概念、大数据思维、大数据基础技术。在读者掌握了这些基础知识之后，再结合大量的代码实例对各个知识点进行深入浅出的讲解，使读者可以在掌握大数据各项技术的基础上，结合实际应用项目进行大数据实践。全书共分为10章，每章的内容简介如下。

第1章：帮助读者更好地认识和了解大数据发展历程，大数据的基本概念及特征，大数据与传统数据在分析和处理方式上的区别，大数据的价值，大数据安全与隐私保护方面的内容。

第2章：介绍要进行大数据实践所涉及的一些关键技术，如大数据采集与预处理技术、大数据存储与管理技术、大数据分析与挖掘技术、大数据应用与展现技术。

第3章：由于Hadoop已成为企业大数据应用的事实标准，因此本章内容以Hadoop为基础，介绍基于Hadoop的大数据生态系统的发展、架构以及构建过程等内容。

第4～9章：分别讲解Hadoop生态系统中各个组件的基本原理、体系构架，以及具体实例等内容。

第10章：介绍如何结合行业特点，利用开源的大数据相关产品，进行大数据产品的设计、开发、部署等过程，并以互联网和智慧交通行业的大数据应用案例为基础，来为读者说明大数据如何展开行业应用以及大数据在行业应用的价值。

另外，本书最后还添加了几个附录，如Hadoop默认端口及作用、Hadoop 1.0与Hadoop 2.0属性名称变化对比、HBase和Hive默认配置说明。本书配套的附件主要由Demo、Hadoop-2.6.0-API、HadoopWorksapces、Softwares和VMwareHadoop文件夹组成。其中，Demo目录中包括HDFS、HBase、YARN、Mahout等Hadoop组件的代码示例；Hadoop-2.6.0-API目录中的index.html可查看Hadoop所有类及功能说明，帮助读

者进行 Hadoop 二次开发；HadoopWorksapces 目录中的每个文件夹都是一个 Eclipse 项目工程，读者将项目导入到 Eclipse 中查看和运行工程；Softwares 目录中包括 Hadoop 生态系统的基本组件、Eclipse IDE 及 Hadoop 的 Eclipse 插件等，方便读者进行搭建大数据平台及二次开发；VMwareHadoop 目录为 Hadoop 集群中的 Master1. Hadoop、Master2. Hadoop、Slave1. Hadoop、Slave2. Hadoop 和 Slave3. Hadoop 节点，读者可通过 VMware 工具直接导入查看已搭建好的 Hadoop 集群。本书的附件可从清华大学出版社官网进行下载。

本书由宋智军编著，同时感谢中国电子科技集团公司第二十八研究所的夏耘、张春晖、高翔、刘文、常庆龙、许家尧、曹博琦、丁旻等给予的帮助和支持。同时非常感谢清华大学出版社的广大员工，他们为本书的选题策划、编辑加工和出版发行付出了辛勤的劳动。由于编者水平有限，加之时间仓促，书中难免有疏漏和不足之处，恳请专家和广大读者指正。

宋智军

目 录

第1章 大数据概述

大数据是继云计算、物联网之后信息技术产业领域的又一重大技术革新。大数据让人们以一种新的数据处理模式对结构化、半结构化以及非结构化的海量数据进行分析，从而获得更强的决策力和洞察力。本章内容旨在帮助读者更好地认识和了解大数据发展历程，大数据的基本概念及特征，大数据与传统数据在分析和处理方式上的区别，大数据现状及发展趋势等。

1.1 大数据发展历程

大数据的概念并不是突然出现的，而是IT技术发展到一定阶段的必然产物。以下是大数据发展过程中一些具有里程碑意义的事件，以及属于大数据概念进化历程中的一些"第一次"或"新发现"等。

早在2008年9月，国际顶级期刊*Nature*就推出了*Big Data*专刊[1]，并邀请一些研究人员和企业家预测大数据所带来的革新。同年，计算社区联盟(Computing Community Consortium)发表了报告*Big-Data Computing：Creating revolutionary breakthroughs in commerce，science，and society*[2]，阐述了在数据驱动的研究背景下，解决大数据问题所需的技术以及在商业、科研和社会领域所面临的一些挑战。

2011年2月，国际顶级期刊*Science*推出*Dealing with Data*专刊[3]，主要围绕着科学研究中大数据的问题展开讨论，专题中的文章既强调了数据洪流所带来的挑战，也强调了如果人们能够更好地组织和访问数据，那么所能抓住和实现的机遇。从而说明大数据对于科学研究的重要性。全球知名的咨询公司麦肯锡(McKinsey)在同年5月份发布了一份关于大数据的详尽报告*Big data：The next frontier for innovation，competition，and productivity*[4]，对大数据的影响、关键技术和应用领域等进行了详细的分析。AMD公司在同年6月份主办的IDC白皮书*Big Data：What It is and Why You Should Care*[5]中，强调了大数据可以用来从海量数据中高效地提取价值，使高速的采集、发现、分析数据成为可能。

2012年1月，达沃斯世界经济论坛特别针对大数据发布了*Big Data，Big Impact：New Possibilities for International Development*报告[6]，该报告重点关注了个人产生的移动数据与其他数据的融合与利用，以及在新的数据产生方式下，如何更好地利用数据来产生良好的社会效益。同年3月，美国二十多位数据管理领域的知名专家从专业的研究

角度出发，经过约三个月的深入研讨，撰写并发布了 *Challenges and Opportunities with Big Data* 白皮书[7]，文章阐述了大数据的产生，分析了大数据处理流水线的各个阶段，指出了其中的诸多技术挑战，并提供了重要的解决思路。与此同时，美国政府发布了 *Big Data Research and Development Initiative*[8]，旨在通过推动和改善与大数据相关的收集、组织和分析工具及技术，提升从海量和复杂的数据集中获取知识和洞察分析能力。美国将大数据作为国家级的战略，其在经济社会发展中的重要地位可见一斑。同年 7 月，联合国的创新倡议项目 Global Pulse 发布了 *Big Data for Development：Opportunities & Challenges* 白皮书[9]，指出大数据促社会发展，对于全世界是一个历史性的机遇，可以利用大数据造福人类。

进入 2013 年，大数据已成为热门话题，并在越来越多的领域当中逐渐得到广泛的应用。实力雄厚的传统 IT 企业及互联网巨头已通过对大数据的存储、挖掘分析、大数据治理等方面进入到大数据领域中掘金。大数据咨询服务 Big Data Group 通过评估上百家提供大数据服务的公司绘制了大数据产业生态地图 2013 版，将大数据产业生态划分为三个层次：大数据应用、大数据基础设施和大数据技术，如图 1.1 所示。

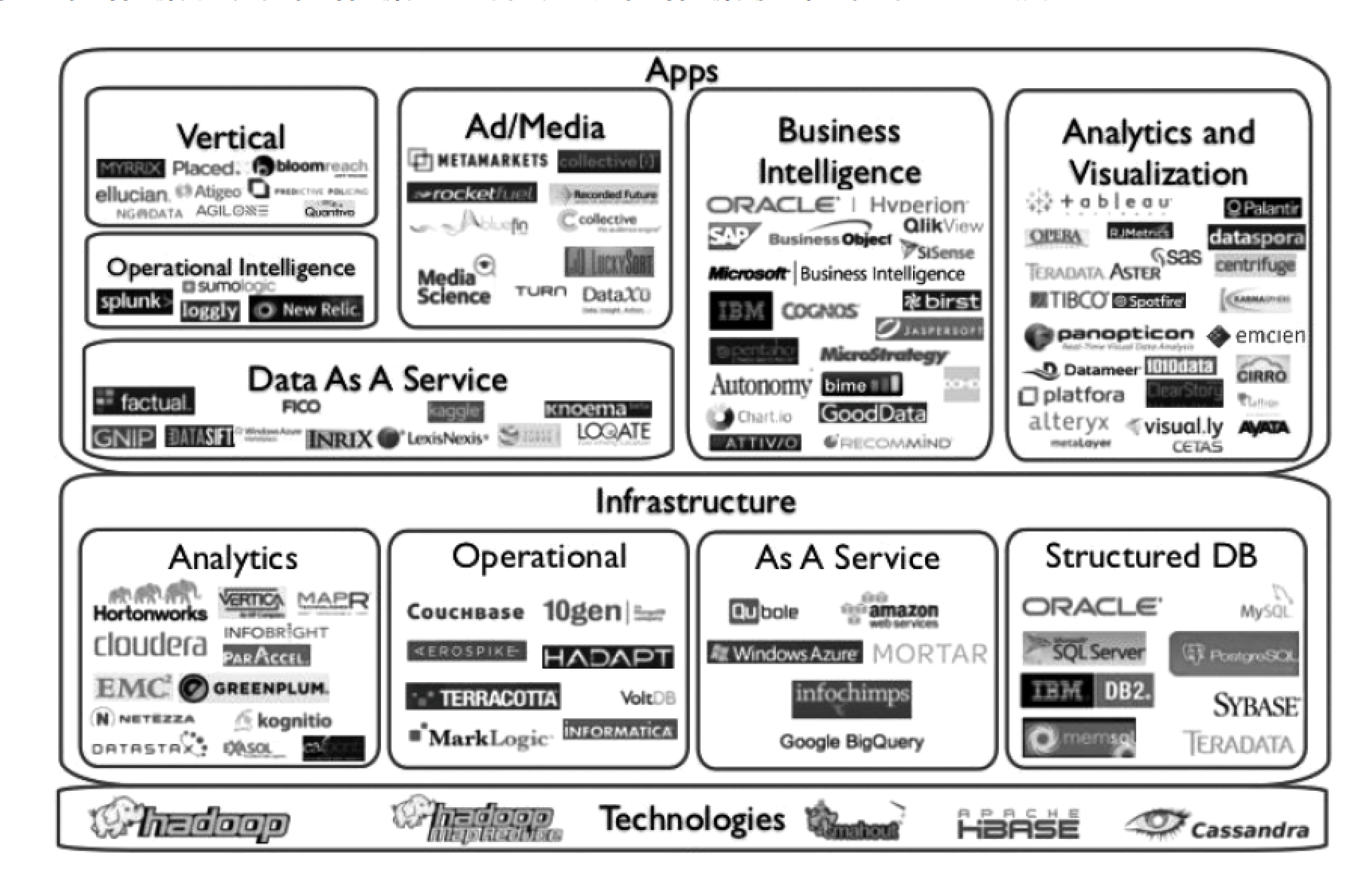


图 1.1　2013 年大数据产业地图

（图片来源：http://www.bigdatalandscape.com/）

1. 大数据应用

目前大数据发展处于初级阶段，大数据应用进程也相对缓慢，主要是互联网领先，其他领域仍在探索中。从图 1.1 中可以看到，一些公司开发出了大数据通用应用，例如大数据可视化和分析工具、大数据商业智能工具或数据服务等。还有一些大数据公司开发出了面向行业用户的垂直应用。未来大数据还将在更多行业得到广泛应用，例如医疗、能源、电信运营、制造业、金融、零售业等。



2. 大数据基础设施

大数据的基础设施不只是简单的物理基础服务器、存储设备等,主要是指大数据平台PaaS层的基础设施,如数据采集、存储、数据集成、数据并行处理和数据分析等基础的平台层能力。

3. 大数据技术

大数据技术包括数据采集、数据存取、数据处理、统计分析、数据挖掘、模型预测、结果呈现等技术。目前,Hadoop已经确立了其作为大数据生态系统基石的地位。Hadoop框架是由Java实现的,它可以对分布式环境下的大数据以一种可靠、高效、可伸缩的方式处理。

到目前为止,大数据市场仍处于初级阶段,也是形成大数据市场竞争格局的关键时期,企业的大数据应用开始从概念验证和实验走向真正的商业化道路,如IBM、Oracle、EMC、SAP等国际IT巨头提供满足客户需求的大数据解决方案,并推出一体化的集成设备等。

1.2 大数据的定义及特征

1.2.1 大数据定义

大数据是一个涵盖多种技术的概念,它的诞生和发展原动力最初来自于互联网的快速发展。然而,对于大数据的定义至今都没有一个被业界广泛采纳的明确定义,可谓是仁者见仁,智者见智。下面给出一些具有代表性的大数据定义。

(1) 麦肯锡全球研究所在其报告 *Big data: The next frontier for innovation, competition, and productivity* 中给出的大数据定义是:大数据指的是大小超出常规的数据库工具获取、存储、管理和分析能力的数据集[4]。但它同时强调,并不是说一定要超过特定TB值的数据集才能算是大数据。

(2) 在维基百科中关于大数据的定义为[10]:大数据指的是所涉及的资料量规模巨大到无法透过目前主流软件工具,在合理时间内达到撷取、管理、处理、并整理成为帮助企业经营决策更积极目的的资讯。笔者认为,这并不是一个精确的定义,因为无法确定主流软件工具的范围,并且可接受时间也是个概略的描述。

(3) 互联网数据中心(IDC)从大数据的4个特征来定义[11],即海量的数据规模(Volume)、数据处理的快速性(Velocity)、多样的数据类型(Variety)、数据价值密度低(Value),即所谓的"4V"特性。然而,IBM认为大数据还应该具有其真实性(Veracity)[12]。

(4) 全球性的信息技术研究和顾问公司Gartner给出了这样的定义[13]:大数据是需要新处理模式才能具有更强的决策力、洞察发现力和流程优化能力的海量、高增长率和多样化的信息资产。

(5) 美国国家标准技术研究院(NIST)将大数据定义为[14]：数量大、获取速度快或形态多样的数据,难以用传统关系型数据分析方法进行有效分析,或者需要大规模的水平扩展才能高效处理。

目前,在对大数据定义的问题上很难达成一个完全的共识,这些定义都是从大数据的特性出发,通过对特性的阐述和归纳来描述大数据。根据大数据的内涵,并结合业界对大数据的理解,在这些定义中,较为普遍认同的是大数据的4V特性。在面对实际问题时,不必过度拘泥于具体的定义之中,在把握4V定义的基础上,适当考虑其他特性即可。

1.2.2 大数据的关键特征

从上述对大数据的定义,提取出大数据的4个关键特征,分别是：数据量大(Volume)、数据类型多(Variety)、处理速度快(Velocity)和数据价值(Value),即大数据的4V性,这些特性使得大数据区别于传统的数据概念。大数据的4V性如图1.2所示。

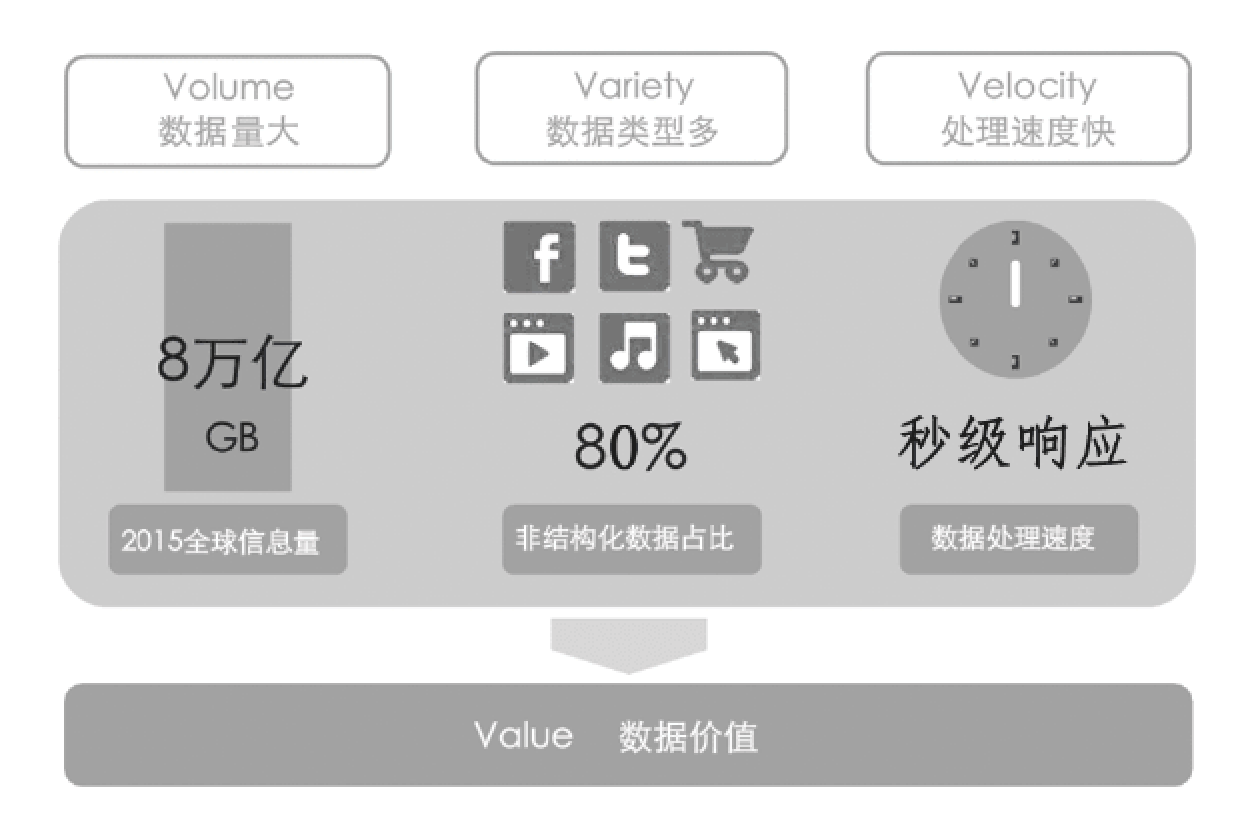


图1.2 大数据的4V性

(图片来源：http://www.iresearch.cn/report/)

注： 初学者很容易对"大数据"与"海量数据"混淆,大数据和海量数据这两个词汇都是来自英文,但翻译不大相同。大数据的英文译文是"big data",直白明了;而海量数据则是"large-scale data"或者"vast data",字面意思很明显,就是规模很大、量很大的数据。海量数据只强调数据的量,而大数据不仅用来描述大量的数据,还更进一步指出数据的复杂形式、数据的快速时间特性以及对数据的分析、处理等专业化处理,最终获得有价值信息的能力。

1. 数据量大

大数据首先是数据量大以及规模的完整性。全球数据量正以前所未有的速度增长,数据的存储容量从TB级别已扩大到ZB数量级。随着传感设备、移动设备、网络宽带的成倍增加,在线交易和社交网络每天生成上百万兆字节的数据,数据规模也在不断地急剧增长。截止到当前,全球的数据总量如图1.3所示。

据EMC公布委托IDC调查的研究报告指出[15],预计在2015年,全球数据总量将会



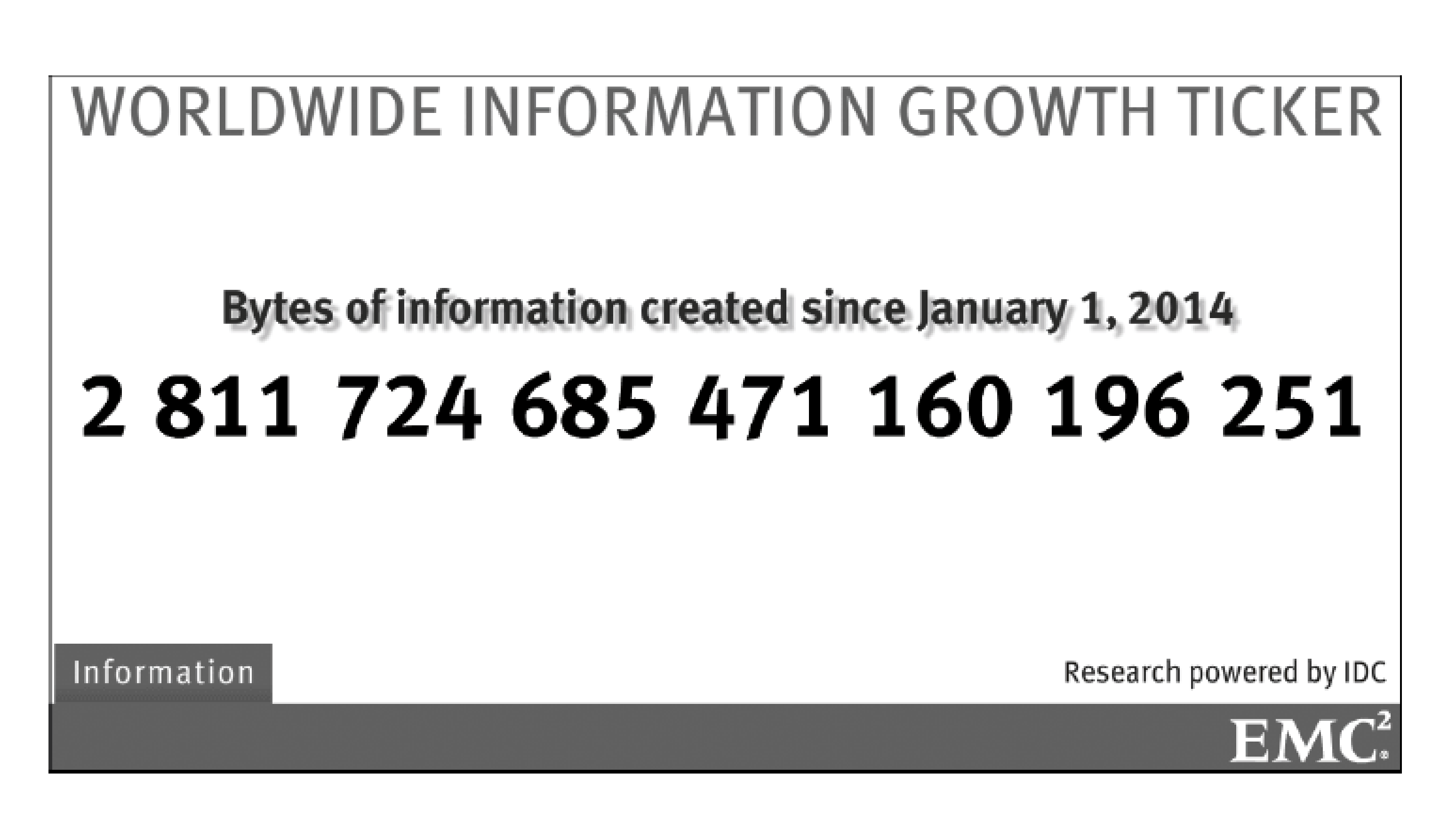


图 1.3　全球数据总量

（图片来源：http://www.emc.com/leadership/programs/digital-universe.htm）

有 8×10^{12}GB 的数据量，到 2020 年，全球将总共拥有 35ZB 的数据量（35ZB 的数据量约等于地球上沙滩上所有沙粒总和的四十多倍），如图 1.4 所示。

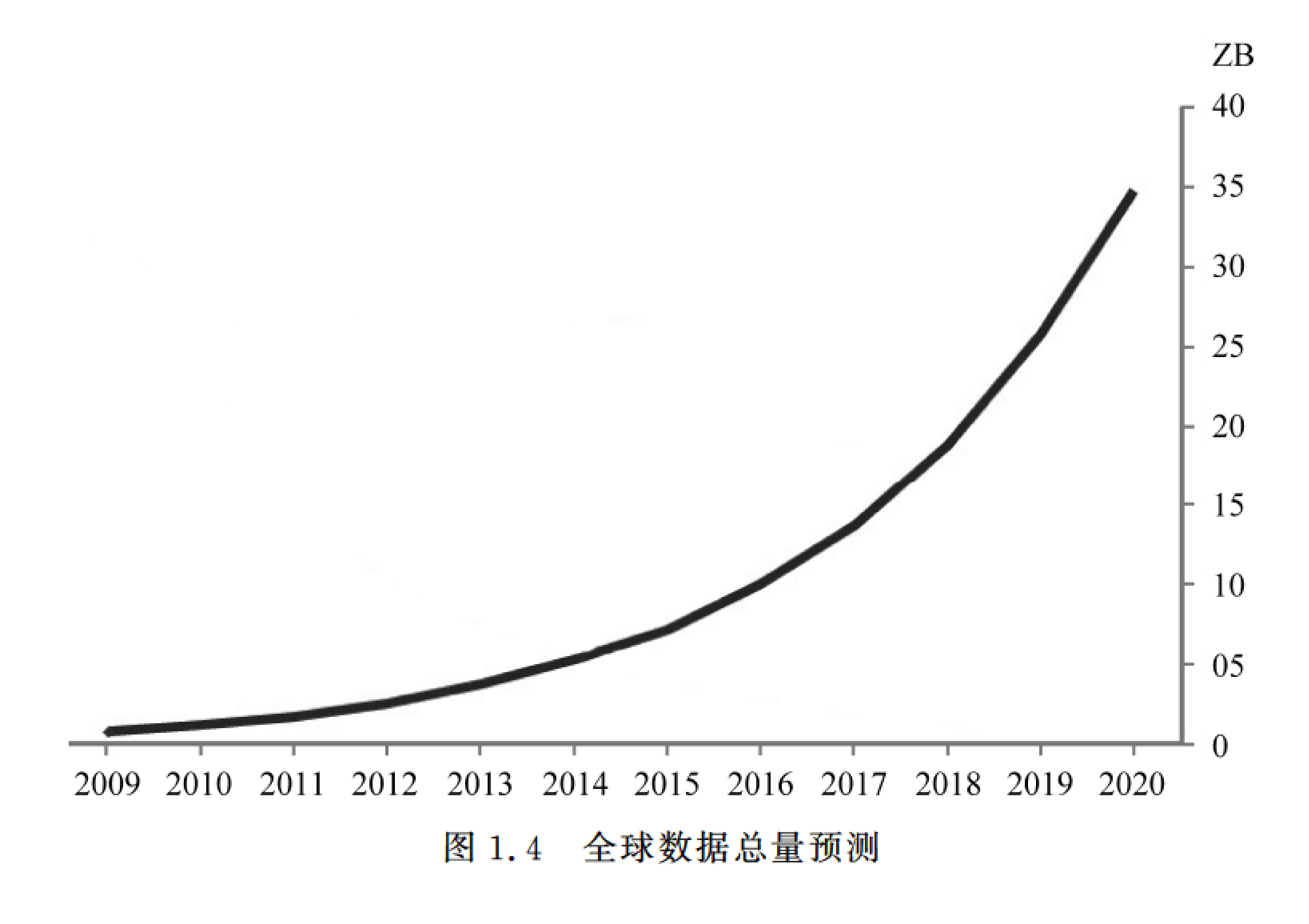


图 1.4　全球数据总量预测

2. 数据类型多

大数据的数据类型非常多。如今的数据类型早已不是单一的以文本为主的结构化数据，随着更多互联网多媒体应用的出现，新型多结构数据量也呈现爆炸式增长，诸如网络日志、电子文档、电子邮件、网页、音频、视频、图片、地理位置信息等大量的非结构化数据已经占到了总数据的很大比重。据 Gartner Group 最新研究报告指出[16]，近两年产生的数据量是过去互联网出现以来所有数据量的总和，而随着社交网络和移动设备的普及，企业 80%的数据是非结构化或半结构化的，结构化数据仅有 20%。同时，全球结构化数据增长速度约为 32%，而非结构化数据的增速则高达 63%。随着非结构化数据的比重越来越大，已逐渐成为大数据的主体。

注：结构化数据是可以用二维表结构来逻辑表达实现的数据，以传统的表格形式存储在数据库或数据仓库中；非结构化数据就是不方便用数据库二维逻辑表来表现的数据，包括所有格式的办公文档、文本、图片、HTML、XML、各类报表、图像、音频、视频等信息。如果把非结构化数据再进行细分，可以分为半结构化数据和非结构化数据两种，半结构化数据就是介于完全结构化数据（如关系型数据库、面向对象数据库中的数据等）和完全无结构的数据（如声音、图像等）之间的数据，HTML、XML、各类报表等就属于半结构化数据；此时的非结构化数据就指完全无结构的数据，如音频、视频等数据。半结构化数据一般是自描述型的，数据的结构和内容混在一起，没有明显的区分。

3. 处理速度快

大数据要求数据处理速度快，是区别于传统数据最显著的特征。现实中则体现在对数据的实时性需求上。如今已是 ZB 时代，在如此海量的数据面前，处理数据的效率就是企业的生命。传统的数据挖掘技术在如此大的数据量下无法快速发现数据规律和有价值信息，必须要求采用大数据的“秒级响应定律”，即海量数据挖掘分析尽可能的秒级响应。否则，再有价值的数据只要过了时效性，也失去了它存在的意义。而对大数据的快速处理分析，将为企业实时洞察市场变化、迅速做出响应、把握市场先机提供决策支持，将成为企业提高竞争力的关键。

4. 价值密度低

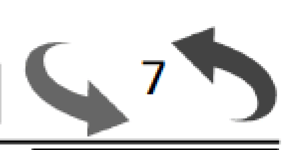
大数据的 3V 构成也导致其数据价值隐藏在海量数据之中，往往表现为数据价值高但价值密度低的特点，需要通过机器学习、统计模型以及图算法等深入、复杂的数据分析才能获得可对未来趋势和模式提供预测性分析的重要洞察力。这些预测性分析要胜过传统商业智能查询和报告的结果。以监控视频为例，连续不间断监控过程中，可能有用的数据仅有一两秒，如何通过强大的机器学习等算法快速、实时地发现价值、提取价值是目前大数据背景下亟待解决的难题之一。

1.3 大数据与传统数据的区别

大数据是在传统数据库学科的分支（数据仓库与数据挖掘）的基础上进一步发展起来的，但两者在数据存储、数据分析、数据处理规模上都有所不同。下面从数据思维、数据处理以及数据分析三方面来介绍两者的不同。

1.3.1 数据思维

大数据思维与传统的数据思维有着很大的差别，传统的数据思维针对一个问题往往是命题假设型的，并通过演绎推理来证明自己的假设是否正确。这种思维方式一般要预先设定好主题，通过建立数据模型和元数据来描述问题。同时，需要理顺逻辑、理解因果关系，并设计算法来得出接近现实的结论。然后，再变更请求重复上述过程（如图 1.5 所示）。例如，某一事件的发生可能是由于 A、B、C 等因素造成，并有一定的理论依据。在这



种情况下，就可以通过传统的数据思维来验证假设是否合理。

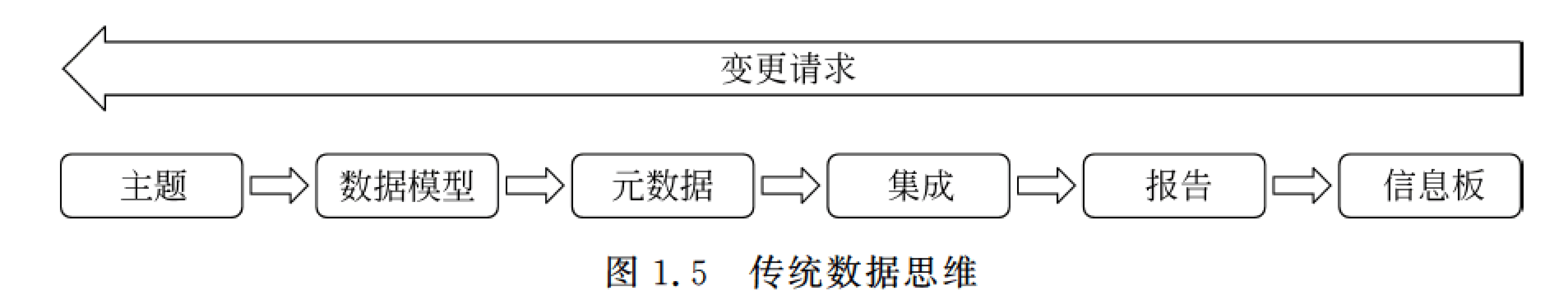


图 1.5　传统数据思维

因此，针对传统的数据思维，一个问题能否得到很好的解决，取决于建模是否合理，各种算法的比拼成为决定成败的关键。当我们有一个明确的问题需要解决时，并针对可能发生的状况有足够的理论支撑时，采用传统的数据思维可以有效地解决该问题。

当我们针对一个没有预制假设的问题时，传统的数据思维就显得无能为力。然而，大数据思维却可以很好地对这类问题进行有效的解决。大数据思维在定义问题时，没有预制的假设，而是使用归纳推理的方法，从部分到整体地进行观察描述，通过问题存在的环境观察和解释现象，从而起到预测的效果(如图 1.6 所示)。例如，把最近几年有关重大交通事故的所有数据收集到一起，对数据所能够揭示出什么有价值的东西没有任何想法时，就可以通过大数据思维进行探索式的发现，从而可以提炼出有价值的信息。

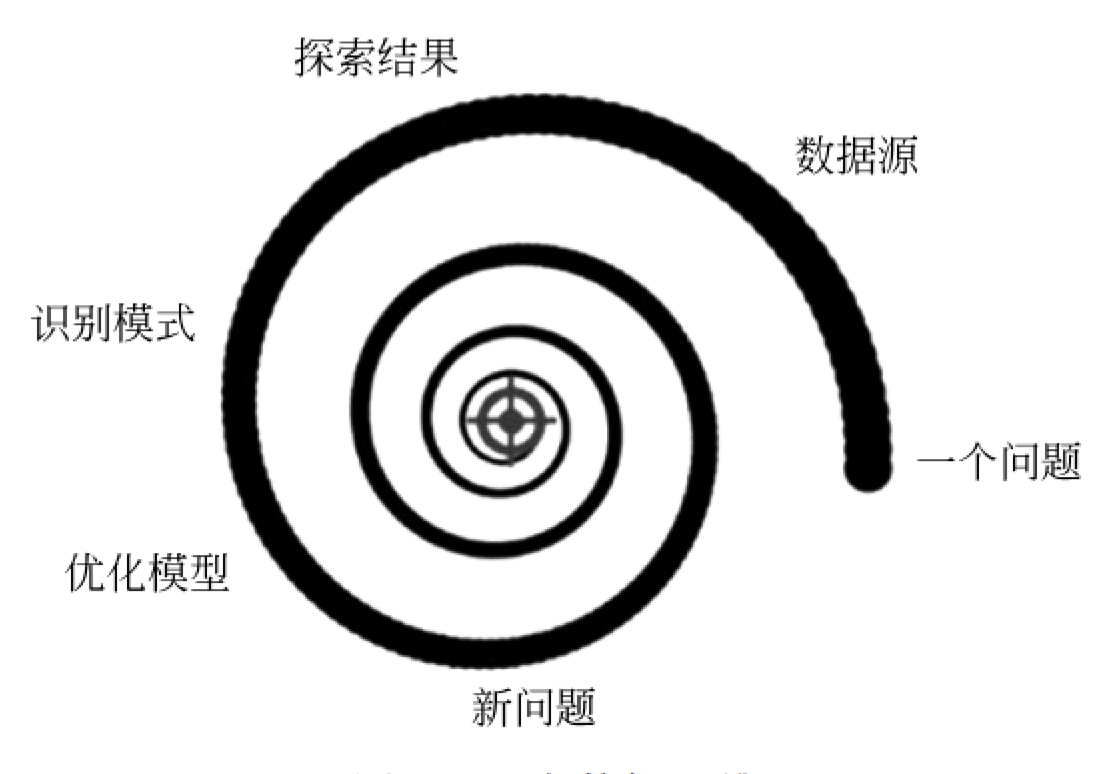


图 1.6　大数据思维

1.3.2　数据处理

传统的数据处理(如图 1.7 所示)主要是面向结构化数据和事务处理的关系型数据库为主，然后通过定向的批处理过程长时间地对数据进行提取、转换和加载(ETL)等处理，处理后的数据是容易理解的、清洗过的，并符合业务的元数据，将这些数据再加载进入数据仓库进行相关性、数据挖掘等分析。对数据的分析是通过昂贵的硬件(小型计算机＋磁盘阵列)或一体机来完成，如大规模并行处理(MPP)系统和/或对称多处理(SMP)系统来实现的。这些硬件平台的兼容性差，扩展性只能达到 TB 级别。

大数据处理技术(如图 1.8 所示)具备结构化、半结构化和非结构化数据混合处理的能力，主要是针对半结构化和非结构化数据。这意味着不能保证输入的数据是完整的，清洗过的和没有任何的错误。这使它更有挑战性，但同时它提供了在数据中获得更多的洞

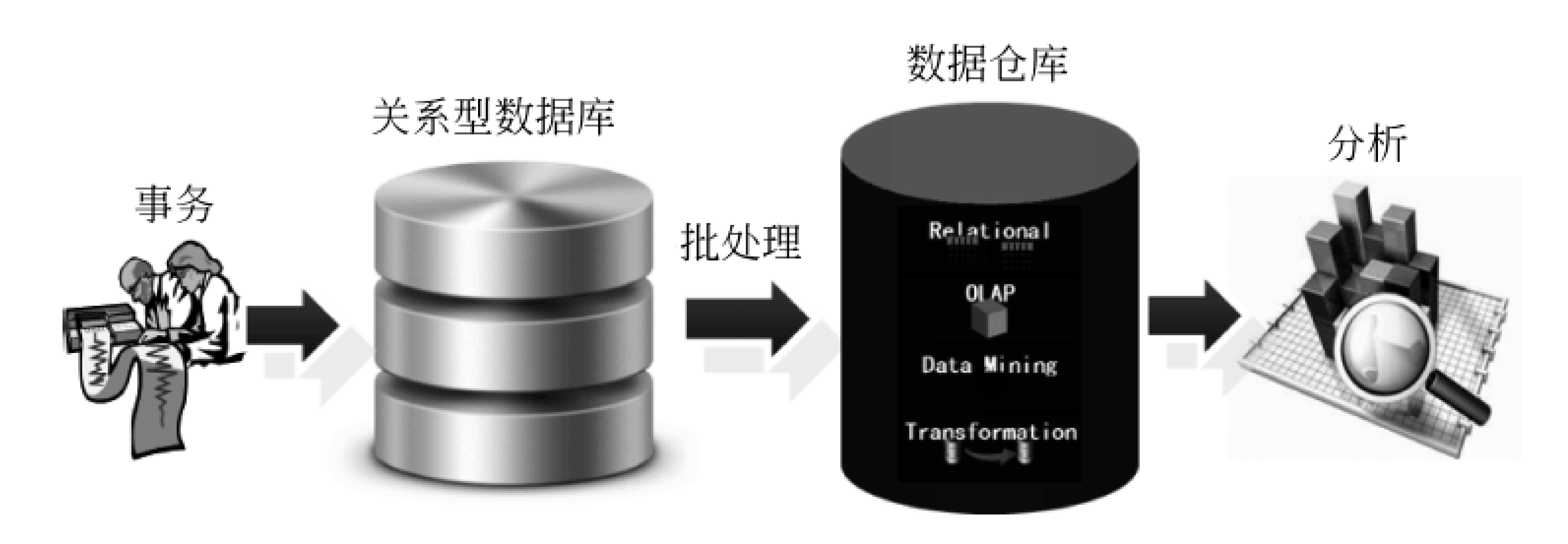


图 1.7　传统数据处理过程

察力的范围。大数据的通用处理过程包括数据的采集、组织、分析和决策。

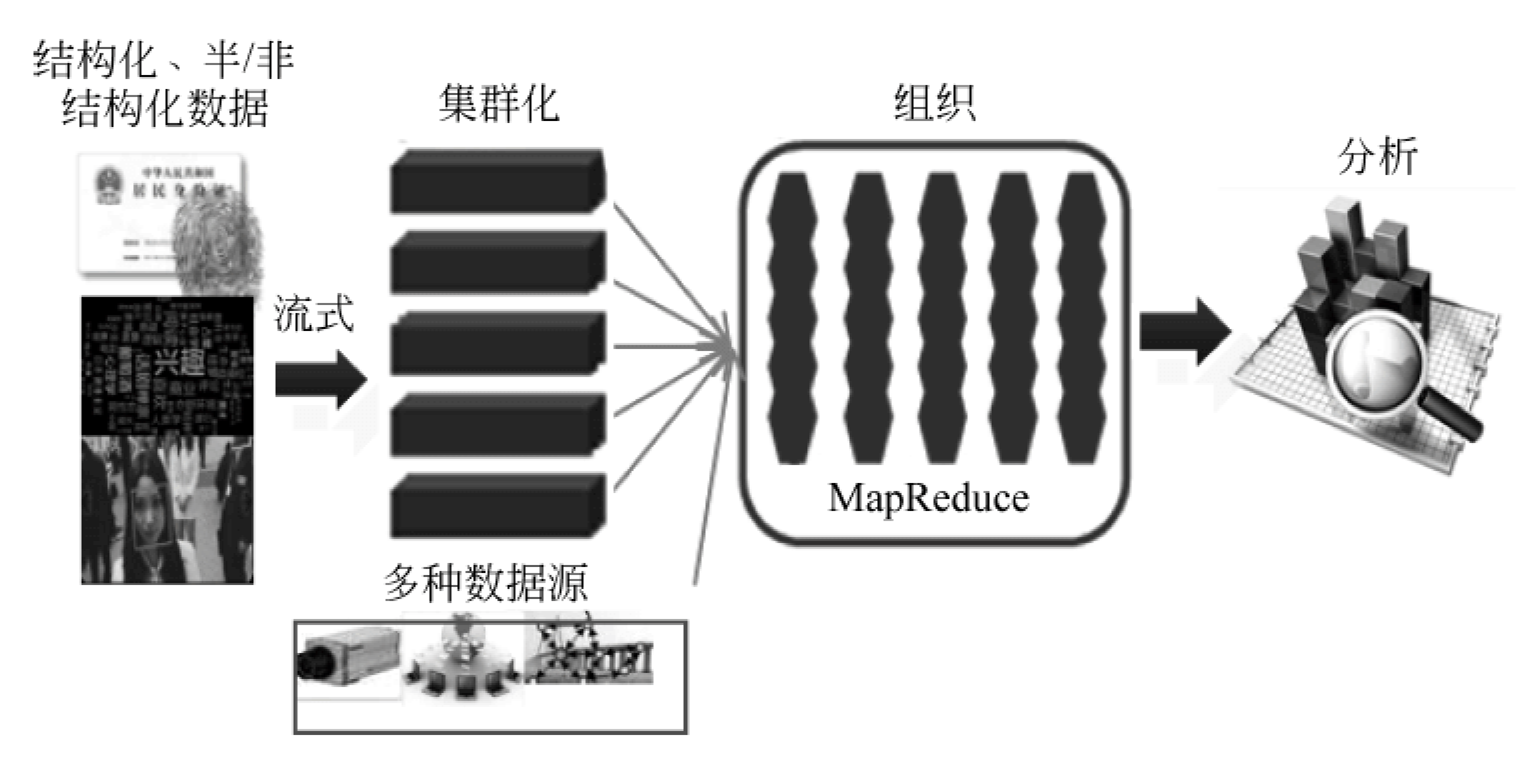


图 1.8　大数据处理过程

在数据采集方面，与传统的数据采集单一来源不同，主要对来源丰富且类型多样的庞大的数据进行采集，并存储在如关系型数据库 Oracle、非关系型数据库 NoSQL、分布式文件系统 HDFS、内存数据库 SAP HANA 等，用户可以通过这些数据来进行实时的查询和处理工作。采集过程的主要特点和挑战是并发数高。

虽然采集端本身会有很多数据库，但是如果要对这些海量数据进行有效的分析，还是应该将这些来自前端的数据导入到一个集中的大型分布式数据库，或者分布式存储集群，并且可以在导入基础上做一些简单的清洗和预处理工作。这些工作可以通过 MapReduce 这一并行处理技术来提高数据的处理速度。

在大数据的分析过程方面，支持针对数据的实时分析、流式分析等。对数据的分析是基于通用硬件，如普通的 x86，并不依赖高性能、高可靠性的硬件，平台兼容性好，可扩展性高，可以达到 PB 级别以上。

在大数据的决策过程方面，就是将在分析中得到的洞察付诸实施。决策过程主要是在现有的数据上进行基于各种算法的计算，从而起到预测的效果，实现一些高级别数据分析的需求、预警、可视化等。

注：MapReduce 是通过大量廉价服务器实现大数据的并行处理，对数据一致性要求不高，其突出优势是具有扩展性和可用性，特别适用于对海量的结构化、半结构化和非结构化数据的混合处理。

1.3.3　数据分析

传统的数据分析通过数据抽样，并不断改进抽样方法来提高样本的精确性，从而对整体数据进行推算，并竭力挖掘数据之间的因果关系；而大数据分析的对象是全体数据，不存在采样的不合理导致预测结果的偏差。传统数据分析的算法比较复杂，通常是用多个变量的方程来追求数据之间的精确关系；而大数据分析则考虑用简单的算法实现规律性的分析。传统的数据分析关注的是“为什么”的因果关系思维方式，而大数据分析关注的是“是什么”的相关性关系，即从海量数据中分析出人类不易感知的关联性。传统数据分析追求的是精确性，即探寻问题的最终答案，而大数据分析是基于海量数据分析得出的结果，该结果一般都是一种供决策参考的指向性意见。下面通过表 1.1 来说明传统的数据分析和大数据分析的区别之处。

表 1.1　传统数据分析和大数据分析

	传统数据分析	大数据分析
分析对象	部分数据的采样	全部数据
分析类型	结构化数据	结构化、半/非结构化数据
精确性	必须接受精确的、规范化的数据	可以是非精确的、非规范化的、不完整的数据
分析算法	对算法的要求比较高	算法简单有效
分析结果	注重因果关系	更注重相关性，而非因果关系

1.4　大数据的核心价值

大数据最有价值的应用就是预测性分析，是以问题为中心，以数据为基础，通过科学地建立模型，进行探索式建模和发现。因此，大数据改变的不仅是技术，更是对业务形态的直接改变，将企业战略从“数据驱动”业务驱动转向“数据驱动”。

例如，互联网企业 Google 通过大数据技术推出了一种名为“谷歌流感趋势”的工具(Google Flu Trend)[17,18]，并且可以通过互联网进行互动查阅。谷歌设计人员认为，人们输入的搜索关键词代表了他们的即时需要，反映出用户情况。为便于建立关联，设计人员编入流感关键词，包括“温度计”、“流感症状”、“肌肉疼痛”、“胸闷”等。只要用户输入这些关键词，系统就会展开跟踪分析，创建地区流感图表和流感地图。全球每星期会有数以百万计的用户在网上搜索健康信息。在流感季节，与流感有关的搜索会明显增多；到了过敏季节，与过敏有关的搜索会显著上升；而到了夏季，与晒伤有关的搜索又会大幅增加。Google 流感趋势会根据汇总的 Google 搜索数据，近乎实时地对全球当前的流感疫情进行估测。为验证“谷歌流感趋势”预警系统的正确性，谷歌多次把测试结果与美国疾病控制和预防中心的报告做比对，证实两者结论存在很大相关性，如图 1.9 所示。

这些图表显示了根据历史查询所得的不同国家和地区的流感估测结果，以及这些结果与官方的流感监测数据的对比。从图中可以看出，根据与流感相关的 Google 搜索查

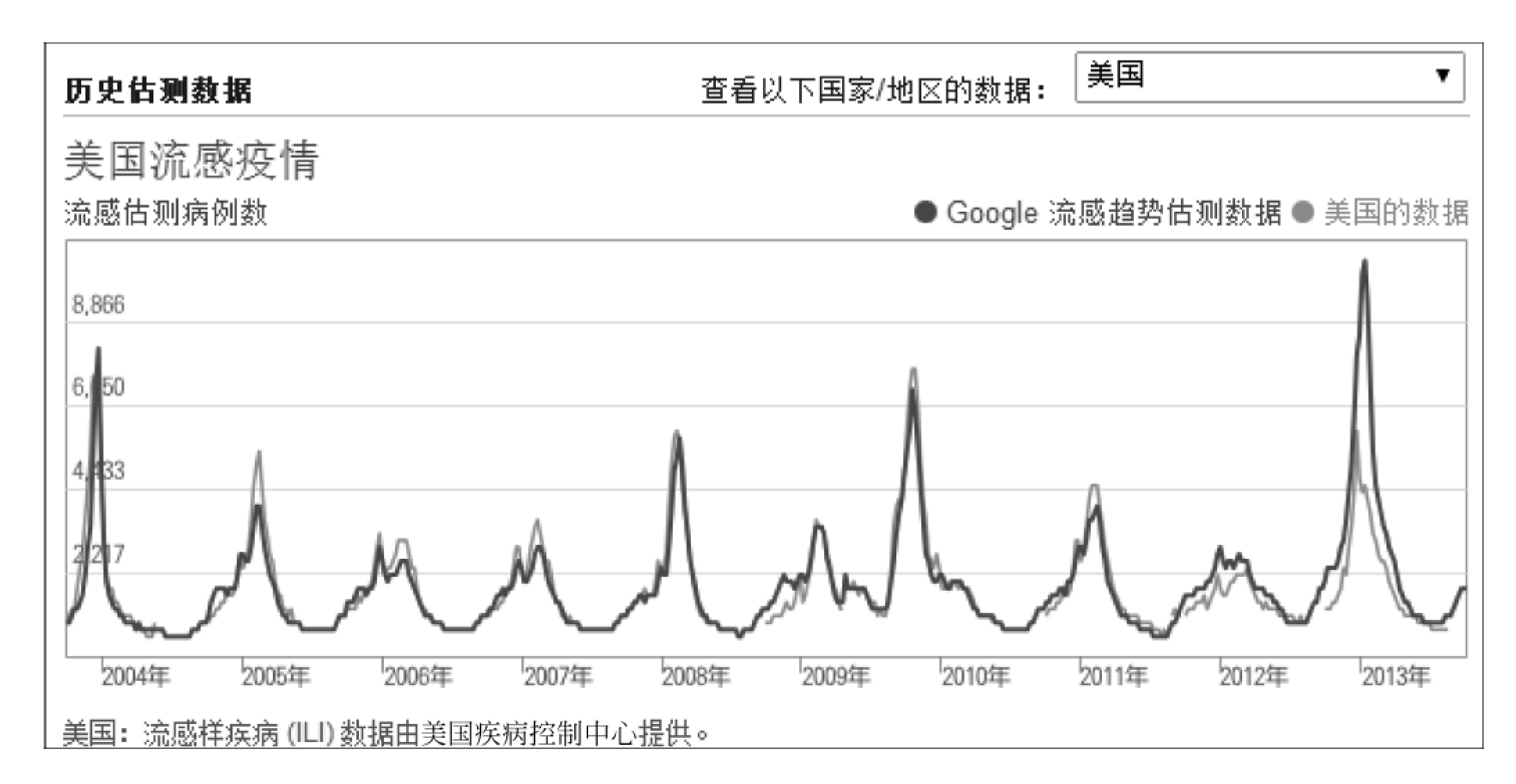


图 1.9 Google 流感估测结果对比

（图片来源：http://www.google.org/flutrends/intl/zh_cn/about/how.html）

询所得到的估测结果，与以往的流感疫情指示线非常接近。当然，过去的表现并不能保证以后的结果一定准确。

这种基于大数据的预测系统的开发能及早监测到疫情暴发的征兆，可以显著减少患病人数。当有新的流感病毒在特定条件下形成后，一旦在全球范围内流行，就可能会夺去数百万人的生命（例如，1918 年就发生过这种灾难）。而通过大数据最新的流感估测结果可帮助相关人员更好地应对季节性流感。

注：对"谷歌流感趋势"的工具（Google Flu Trend）感兴趣的读者，可以阅读美国《自然》杂志发表的文章"运用搜索引擎查询数据检测流感疫情"（*Detecting influenza epidemics using search engine query data*）。

大数据在互联网方面可以对消费者的行为进行预测。据《纽约时报》首席撰稿人 Charles Duhigg 的一份关于《公司如何掌握你的秘密》（*How Companies Learn Your Secrets*）中写道[19]，"据杜克大学研究显示，是习惯而非有意识的决策促成了我们每天 45% 的选择（One study from Duke University estimated that habits, rather than conscious decision-making, shape 45 percent of the choices we make every day）。"只要了解习惯的形成方式就可以更简单地控制它们。通过数据分析消费者的行为便能准确地预测下一步的消费，例如国内的百度预测。百度大数据部通过对搜索数据的深度挖掘，发现旅游相关词搜索数量和实际旅游人数之间的密切关系，并建立了旅游预测模型。该模型可以反映各旅游景点未来的人流趋势，可以对旅游行业进行宏观把握和调控，并可以对目的地营销活动进行引导、对旅游人流流向和流量进行调整，如图 1.10 所示。

大数据的预测性价值为互联网、电信、金融、医疗、制造、流通等提供了一个前所未有的机会，用先前不能做到的方式在感知风险、生成洞察、帮助辅助决策、提高社会安全等方面起到了不可估量的作用。



图 1.10　百度旅游预测

（图片来源：http://trends.baidu.com/tour/）

1.5　大数据安全与隐私保护

大数据正影响着人们日常生活和工作习惯，但随着数据产生的速率、数据来源和格式的多样性、数据高容量存储等特性，数据在传输、存储、管理、数据分析和数据挖掘等过程中存在着诸多安全风险和隐私保护问题。因此，大数据时代的数据安全与隐私保护对传统的安全机制提出了新的挑战，同时也为信息安全领域带来了新的发展契机。传统的数据安全与隐私保护机制是针对小规模的、静态的数据而定制的，对于大数据来说并不适合。因此，与传统的信息安全相比，大数据安全与隐私保护面临的主要挑战体现在以下几个方面。

1.5.1　基础设施安全

基础设施安全包括支持大数据存储和管理的公共基础设施、访问和使用大数据的并行计算、数据挖掘等工具，如硬件设备、云存储、NoSQL、Hadoop、MapReduce 等。下面以 NoSQL 为例简要分析一下 NoSQL 在基础设施安全方面存在的漏洞。

NoSQL 数据库主要提供非关系型数据存储的安全基础设施，用来存储和处理海量的静态数据和流数据。NoSQL 与传统的关系型数据库相比，最显著的特点是性能和可扩展性，同时也引来了很多 NoSQL 的关键性安全漏洞，如只能保证事务完整性的软方法、弱身份认证机制、低效率授权机制、缺乏一致性等安全漏洞。

1. 无法保证事务完整性

NoSQL 无法像传统的关系型数据库一样保证事务的完整性，但提供了一种事务完整性的架构权衡分析的软方法，能在性能和安全方面做出权衡和取舍，使得体系结构在牺牲

安全的前提下可以不引入复杂的完整性约束，从而实现更好的性能和可扩展性。

2. 弱身份认证机制

NoSQL 使用弱认证技术和弱密码存储机制，这种弱机制容易遭到各种注入攻击，攻击者可以利用架构注入大量数据到数据库，从而导致数据库中的数据损坏，使得整个数据库不可用，如 JSON 注入、视图注入、REST 注入、模式注入等。

3. 低效率授权机制

NoSQL 解决方案的授权机制各不相同，大多数流行的解决方案是采用高层次执行授权，而不是在较低层执行，即授权是在数据库级别，而不是集合级别执行，没有在架构中集成基于角色的访问控制机制。

4. 缺乏一致性

NoSQL 并不遵循 CAP 定理（一致性、可用性和可分区容忍性），在任意给定时间用户不能保证一致的结果，因为每个参与节点可能没有跟保持最新数据的节点完全同步，从而有可能导致群集节点之间的负载均衡。

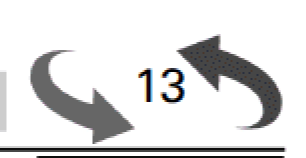
NoSQL 目前的关键性安全漏洞主要是因为 NoSQL 的安全并没有在设计阶段处理，只是由 NoSQL 数据库的开发人员在中间件中嵌入安全，只有一个非常薄的安全层，而且依赖于外部执行机制所造成的。因此，在设计 NoSQL 时应考虑到数据的完整性可以通过应用程序或中间件层实施；数据存储方面应该考虑不使用明文进行传输，而使用加密或使用安全散列算法等；身份认证方面应该在已采用的技术基础上基于硬件设备的加密/解密速度更快；在授权机制方面应该考虑支持可插拔认证模块以具备环境需求的所有级别的安全实施。

1.5.2 数据隐私

当人们在感受大数据所带来的巨大应用潜力和空间的同时，也带来了新的数据隐私问题。大数据潜在地允许隐私侵犯、侵袭式营销、用户消费习惯挖掘、群组特征发现等行为，这使得数据隐私问题越来越严重。如果仅仅为了保护隐私就将所有的数据都加以隐藏，那么又无法体现数据的真正价值；而数据公开必然会对数据隐私造成一定的影响。因此，在大数据时代，数据隐私是无法回避的现实问题。我们只能通过技术手段和建立隐私机制，尽量防止意外的隐私披露。

1. 隐私保护机制

目前用户数据的收集、存储、管理和使用均缺乏规范的管理和监管，主要依靠数据拥有者的自律，而且用户无法确定自己的隐私数据用途，无法决定自己的数据是否能被公开，何时能被销毁等。可以通过建立完善的数据隐私保护机制，来使用户可以决定自己的数据是否要隐私保护，是否对隐私数据进行销毁等操作，如建立数据采集时的隐私保护、数据共享和公开时的隐私保护、数据分析时的隐私保护、数据生命周期的隐私保护、隐私



数据可信销毁等机制。

2. 数据加密技术

数据隐私除应遵循用户隐私法规和隐私安全机制外，还应该结合密码学技术对数据本身进行加密来实现对用户隐私的保护，如对数据本身进行加密封装、从源头去限定数据的可见性、通过限制访问底层系统来控制数据的可见性、数据匿名保护技术、保护隐私的数据挖掘技术等。

3. 访问控制

目前的访问控制是以角色为基础采用自顶向下的管理模式进行访问控制，这种访问控制通过为用户指派角色，将角色关联至权限集合，当其应用于大数据场景时，面临需大量人工参与角色划分、授权的问题等。因此，在访问控制方面，可以考虑使用自底向上的细粒度访问控制方式，首先跟踪单个数据元素的保密要求，当有多个不同的应用共享环境时，要标注该数据元素的最小上界；其次，跟踪用户角色和权限，如果该用户被正确验证，可从多个可信来源数据中提取该用户的安全属性，从而可以妥善处理联合授权；最后，正确实施访问控制的保密要求，它包含要求的访问数据和用户连接属性等。

1.5.3 数据治理

数据治理是用于整个数据生态系统的全面方法，是大数据技术的重要环节。大数据的数据治理比传统结构化数据的数据治理难度更大，既有基于传统的编程模式，又有大数据下特有的方式，无法利用传统数据治理的方式来操作。大数据的数据治理不是用于大数据的局部解决方案，而是贯穿于整个大数据的生命周期的系统全面的方法。因此，大数据的数据治理不能在拥有大量数据之后才考虑数据治理，而是在数据开始生成之前就应该考虑用大数据的方式来进行数据治理。大数据的数据治理包含隐私性、安全性、合规性、数据质量、元数据管理、主数据管理以及结合业务特点的延伸等。企业在进行大数据的数据治理时，应把数据作为宝贵的企业资产来进行管理。此外，还应注意以下几点。

1. 数据维度化

在大数据进行数据治理前，应先考虑对数据进行维度化，即要加强大数据应用的安全起源，明确数据认责，如数据起源记录可靠、隐私保护、可访问控制、数据分类、责任机制等。其中，明确数据的责任方是数据治理的重要方面，只有建立健全的数据认责机制，才能对数据安全、数据质量和数据效率进行持续改进。

2. 数据隐私

大数据进行分析的数据集往往包括个人和组织的隐私信息，而数据隐私也是数据治理的最重要的一方面。数据隐私使得企业承载了更多的责任和最大的业务风险，侵犯个人或组织隐私不仅可能损害企业的声誉、削弱市场信任，更有可能被法律制裁。因此，我们要强制访问控制的保密要求，适当地屏蔽个人或组织隐私信息等措施。

3. 治理目标

大数据的数据治理的目的不是为了大数据分析处理,而是出于对商业决策起指导作用。因此,明确企业的数据治理目标也同样至关重要。针对不同的企业,治理目标不同,如可以对数据是否有明确定义、明确的责任方、数据内容是否符合标准要求、数据的存储与管理、数据分析、数据访问安全控制等方面进行制定。

1.5.4 被动安全机制

大数据安全不能只局限于大数据基础设施的方案,还要考虑到数据采集、大数据分析、大数据基础架构本身的安全。因为大数据需要从各种来源进行数据采集,而哪些数据是我们能信任的数据,哪些数据需要验证,哪些数据是恶意攻击数据,以及大数据基础设施的节点是否安全等,面对这些影响大数据安全的情况,可以采用以下方式。

1. 采集端验证/过滤

大数据的数据来源主要是从采集端进行数据采集,而数据是否可信、是否恶意攻击等就需要在数据采集端对数据进行输入验证和过滤,从而防止入侵者生成并发送恶意数据或对数据进行篡改的恶意行为进行有效的检测和过滤。由于现实世界的数据采集系统收集的数据都是以百万量来计算的,单一的数据输入验证和过滤技术很难满足实际应用。因此,我们建议在实践中使用一种混合的方法来实现数据采集。

2. 实时安全监控

大数据可以带来潜在的价值、有用的辅助决策以及实时的数据处理等,然而这一切实现的前提条件是要保证真实的数据来源,稳定的大数据基础架构,安全的大数据分析等。因此,大数据最具挑战的问题之一就是如何实现实时安全监控,监控大数据基础架构本身,监控大数据基础设施的所有节点的性能和健康情况,监控大数据平台是否受到攻击并实时预警等。目前主流的大数据 Hadoop 平台中并没有内置安全监控和分析工具,这就要求我们在使用 Hadoop 平台时在注重平台稳定性的基础上,进行实时安全监控解决方案和框架的开发。

第2章 大数据关键技术

大数据技术就是从各种类型的海量数据中快速获得有价值信息的技术。根据大数据处理的生命周期，大数据技术通常包括大数据采集与预处理，大数据存储与管理，大数据分析与挖掘，大数据应用与展现。要真正地从大数据中获得洞察，需要在大数据生命周期的各个阶段中全面集成可与之相互配合的技术。本章内容旨在帮助读者更好地认识和了解大数据生命周期各个阶段的关键技术。

2.1 大数据采集与预处理技术

在大数据时代，谁掌握了数据，谁就有可能掌握未来，而其中数据采集和预处理是大数据价值挖掘最重要的一环，其后的数据分析与挖掘、数据管理都构建于采集的基础上。一般来说，在进行数据存储和处理之前，需要对数据进行清洗、整理，这个环节在传统的数据处理过程中其实就是对数据进行采集和预处理。传统的数据分析可以先通过 ETL 工具(Extract-Transform-Load，抽取-转换-装载)进行数据采集和预处理，然后再进行如离线统计分析、机器学习、搜索引擎的计算等。但面对大数据时，传统的 ETL 工具将无法发挥其作用，因为大数据的数据来源复杂多样，如网络日志、视频、图片、地理位置、传感器数据、物联网数据等，而且这些数据格式多样、数据量大，格式的转换开销过大，以及在性能上无法满足海量数据的采集需求。因此，大数据的采集与预处理技术就要实现利用多个数据库来接收来自客户端(Web、App 或者传感器形式等)的数据，并且应该将这些来自前端的数据导入到一个集中的大型分布式数据库，或者分布式存储集群，同时可以在导入基础上做一些简单的清洗和预处理工作。

在大数据的数据采集和预处理过程中，其主要特点和挑战是数据量、数据质量和采集性能。因为在大数据环境下，数据来源多，而且类型也多种多样，采集速度又快，如果不能及时对数据质量进行处理，就会导致数据质量问题的堆积，从而使得质量问题越来越严重。例如，淘宝自主研发的 Time Tunnel 数据采集工具每天要实时采集来自淘宝主站的用户、店铺、商品和交易等数据库的数据，还有用户的浏览、搜索等行为日志等上百万的数据量，如果无法实现对数据质量的实时监控和数据清洗工作，以及在采集性能上无法满足业务需求的话，将无法实现实时日志收集、数据实时监控、广告效果实时反馈、数据库实时同步等。所以在数据采集和预处理环节，如何进行实时数据质量监控和清洗，如何通过强大的集群和分布式计算能力提高数据质量监控性能，如何保证负载均衡和高可靠性等方

面都是需要深入思考和设计的。

目前，大数据的采集工具有 Cloudera 公司的 Flume[21]、Facebook 公司的 Scribe[22]、LinkedIn 的 Kafka[23]、淘宝的 Time Tunnel[24] 以及开源社区 Hadoop 的 Chukwa[25] 等，这些大数据采集工具均可以满足每秒数百 MB 的日志数据采集和传输需求。

2.1.1 Flume

Flume[21] 是一个高可用的，高可靠的，分布式的海量日志采集、聚合和传输的系统。Flume 支持在日志系统中定制各类数据发送方，用于收集数据；同时，Flume 提供对数据进行简单处理，并写到各种数据接受方(可定制)的能力。初始的 Flume 版本是 Flume OG(Flume Original Generation)由 Cloudera 公司开发，叫做 Cloudera Flume；后来，Cloudera 把 Flume 贡献给 Apache，版本改为 Flume NG(Flume Next Generation)现在称为 Apache Flume。

Flume 对日志数据的收集通过三种节点：Master、Collector 和 Agent。Master 是管理节点，用于管理协调 Agent 和 Collector 的配置等信息；Agent、Collector 都属于日志收集节点，Agent 用于采集数据，是 Flume 中产生数据流的地方，同时，Agent 会将产生的数据流传输到 Collector 节点；Collector 用于将多个 Agent 的数据汇总后加载到存储系统中(存储系统可以是普通的文件，也可以是 HDFS、HIVE、HBase 等)。Flume 中还有个重要的概念叫数据流(Data Flow)，即数据传输管道，描述了日志数据从产生处到最终目的地的数据传送过程。同时，数据的传输需要指定数据源(Source)和数据汇集点(Sink)，通过对日志收集节点的 Source，Sink 配置，实现数据流的建立。Flume 的逻辑架构如图 2.1 所示。

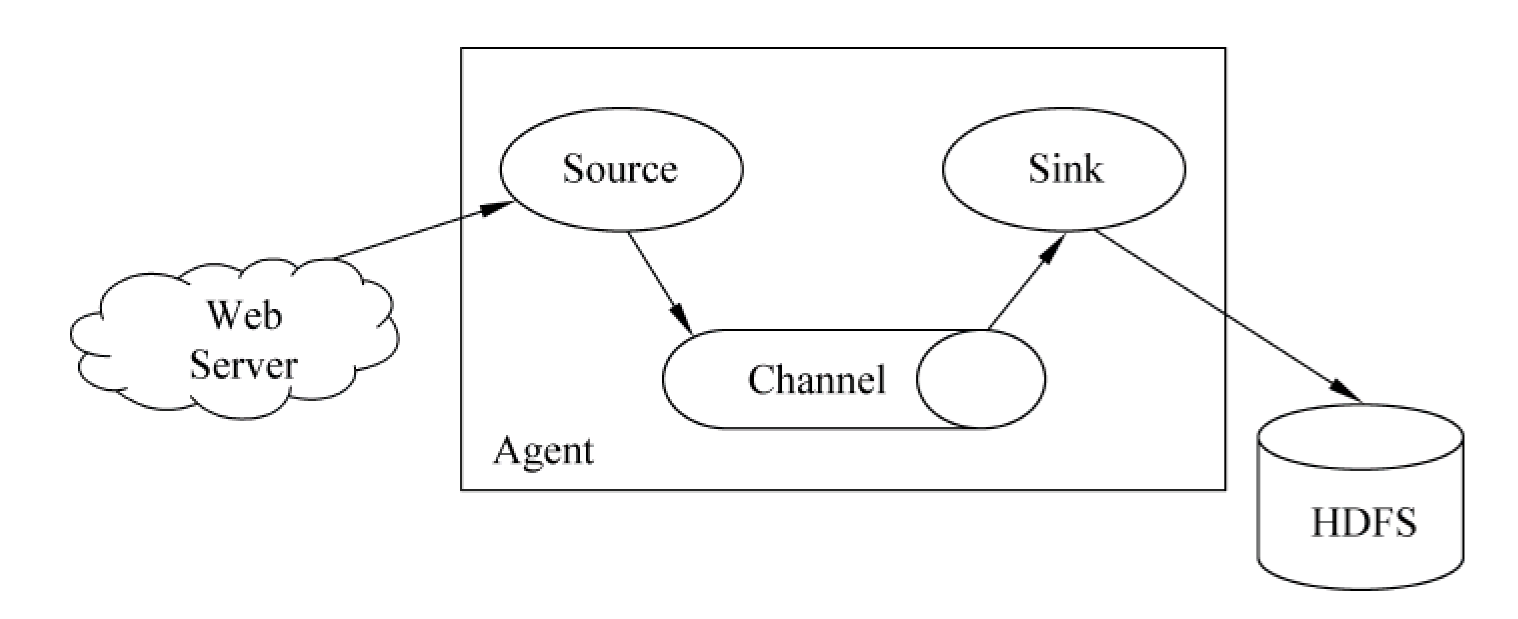


图 2.1 Flume 的逻辑架构

(图片来源：http://flume.apache.org/)

Flume 提供了从 console(控制台)、RPC(Thrift-RPC)、text(文件)、tail(UNIX tail)、syslog(syslog 日志系统，支持 TCP 和 UDP 等两种模式)，exec(命令执行)等数据源上收集数据的能力。同时，Flume 的数据接收方，可以是 console(控制台)、text(文件)、dfs(HDFS 文件)、RPC(Thrift-RPC)和 syslog TCP(TCP syslog 日志系统)等。其中，收集数据有如下两种主要工作模式。

(1) Push Sources：外部系统会主动地将数据推送到 Flume 中，如 RPC、syslog。

(2) Polling Sources：Flume 到外部系统中获取数据，一般使用轮询的方式，如 text

和 exec。

下面简要分析一下 Flume 在可靠性、可扩展性、可管理性及功能可扩展性方面的能力。

1. 可靠性

当节点出现故障时，日志能够被传送到其他节点上而不会丢失。Flume 提供了三种级别的可靠性保障，从强到弱依次分别为：end-to-end（收到数据 Agent 首先将 event 写到磁盘上，当数据传送成功后，再删除；如果数据发送失败，可以重新发送）；Store on failure（这也是 scribe 采用的策略，当数据接收方 crash 时，将数据写到本地，待恢复后，继续发送）；Best effort（数据发送到接收方后，不会进行确认）。

2. 可扩展性

Flume 采用了三层架构，分别为 Agent，Collector 和 Storage，每一层均可以水平扩展。其中，所有 Agent 和 Collector 由 Master 统一管理，这使得系统容易监控和维护，且 Master 允许有多个（使用 ZooKeeper 进行管理和负载均衡），这就避免了单点故障问题。

3. 可管理性

所有 Agent 和 Collector 由 Master 统一管理，这使得系统便于维护。在多 Master 情况下，Flume 利用 ZooKeeper 和 Gossip 协议，保证动态配置数据的一致性。用户可以在 Master 上查看各个数据源或者数据流执行情况，且可以对各个数据源配置和动态加载。其中，Flume 提供了 Web 和 Shell Script Command 两种形式对数据流进行管理。

4. 功能可扩展性

用户可以根据需要添加自己的 Agent，Collector 或者 Storage。此外，Flume 自带了很多组件，包括各种 Agent（file，syslog 等），Collector 和 Storage（file，HDFS 等）。

Flume 本身具有可靠性、可扩展性、可管理性、功能可扩展性，适用于实时的日志收集，安装简单，但动态配置复杂。当系统集群减少时，可通过停止 Flume 日志收集节点的方式进行调整。但是当系统节点增加时，需要用户手动配置新的日志收集节点。配置的过程需要用户对原有系统、Flume 基本概念和配置过程以及 ZooKeeper 有一定程度的了解，所以对用户的要求比较高。

注：Flume 框架对 Hadoop 和 ZooKeeper 的依赖只是在 jar 包上，并不要求 Flume 启动时必须将 Hadoop 和 ZooKeeper 服务也启动。

2.1.2　Scribe

Scribe[22] 是 Facebook 开源的日志收集系统，在 Facebook 内部大量使用。它能从各种日志源收集日志，存储到一个中央存储系统上，便于进行集中统计分析处理。它为日志的“分布式收集，统一处理”提供了一个可扩展的，高容错的方案。它最重要的特点就是容

错性好，Scribe从各种数据源上收集数据，放到一个共享队列上，然后Push到后端的中央存储系统上。当中央存储系统出现故障时，Scribe可以暂时把日志写到本地文件中，待中央存储系统恢复性能后，Scribe把本地日志续传到中央存储系统上。Scribe的逻辑架构如图2.2所示。

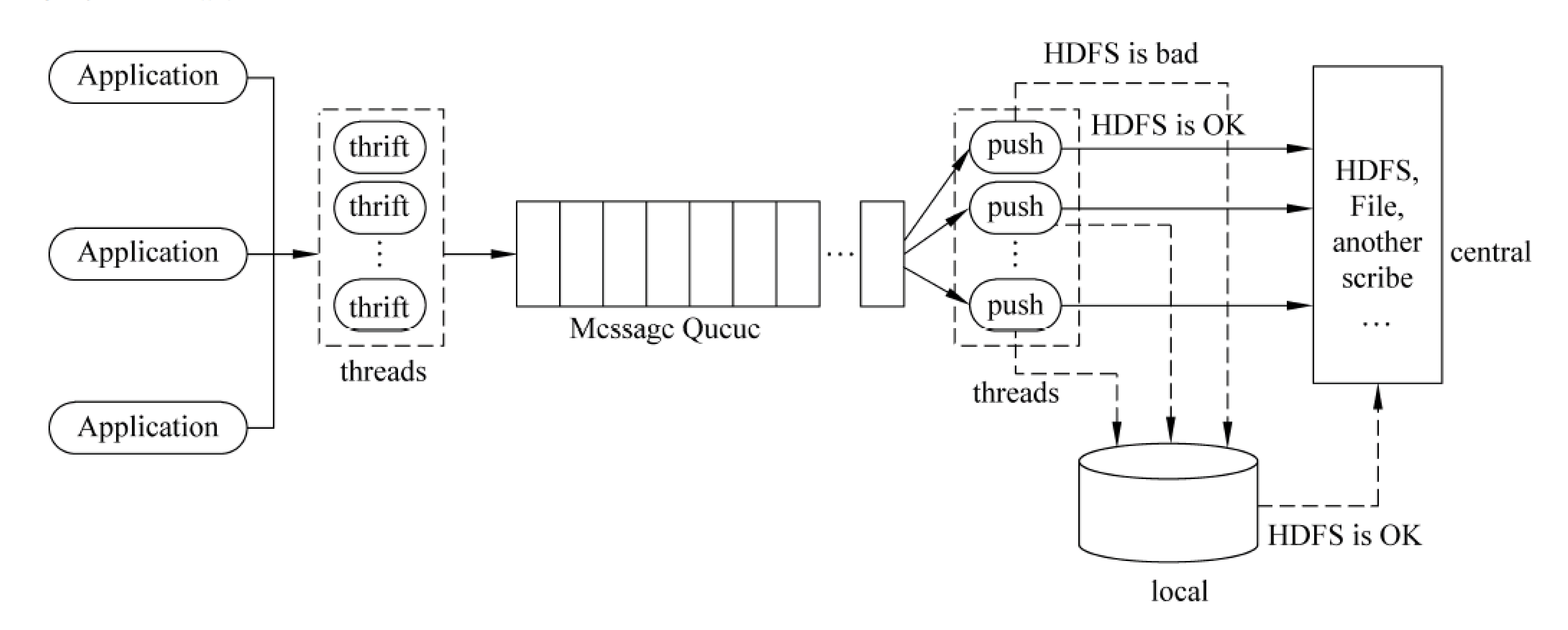


图2.2　Scribe的逻辑架构

（图片来源：http://dongxicheng.org/search-engine/scribe-intro/）

需要注意的是，各个数据源须通过Thrift（由于采用了Thrift，客户端可以采用各种语言编写）向Scribe传输数据（每条数据记录包含一个Category和一个Message）。可以在Scribe配置用于监听端口的Thrift线程数（默认为3）。在后端，Scribe可以将不同Category的数据存放到不同目录中，以便于进行分别处理。后端的日志存储方式可以是各种各样的存储，包括File、Buffer、Network（另一个Scribe服务器）、Bucket（包含多个存储，通过Hash将数据存到不同Store中）、Null、Thriftfile（写到一个Thrift TFileTransport文件中）和Multi（把数据同时存放到不同存储中）。

Scribe为日志收集提供了一种容错且可扩展的方案。Scribe可以从不同数据源，不同机器上收集日志，然后将它们存入一个中央存储系统，以便于进一步处理。当采用HDFS作为中央系统时，可以进一步利用Hadoop进行处理数据，于是Scribe＋HDFS＋MapReduce方案便诞生了。

注：

（1）File：将日志写到文件或者NFS中，目前支持两种文件格式，即STD和HDFS，分别表示普通文本文件和HDFS。

（2）Buffer：这是最常用的一种存储，该存储中包含两个子存储，其中一个是主存储，另一个是辅存储。日志会优先写到主存储中，如果主存储出现故障，则Scribe会将日志暂存到辅存储中，待主存储恢复性能后，再将辅存储中的数据复制到主存储中。其中，辅存储仅支持两种存储方式，一个是File，另一个是HDFS。

（3）Null：这也是一种常用的存储。用户可以在配置文件中配置一种叫Default的Category，如果数据所属的Category没有在配置文件中设置相应的存储方式，则该数据会被当作Default。如果用户想忽略这样的数据，可以将它放入Null存储中。

2.1.3 Kafka

Kafka 是 LinkedIn 公司在 2010 年开发的分布式消息订阅发布系统，用 Scala 开发，目前已经开源并贡献给 Apache，并成为 Apache 的顶级项目。它原本用作 LinkedIn 的活动流(Activity Stream)[①] 和运营数据处理管道(Pipeline)的基础，以实时处理消息，低 I/O 消耗见长，因此多用于大数据实时处理和离线消息处理。现在它已为多家不同类型的公司作为多种类型的数据管道(Data Pipeline)和消息系统使用。Kafka 的逻辑架构如图 2.3 所示。

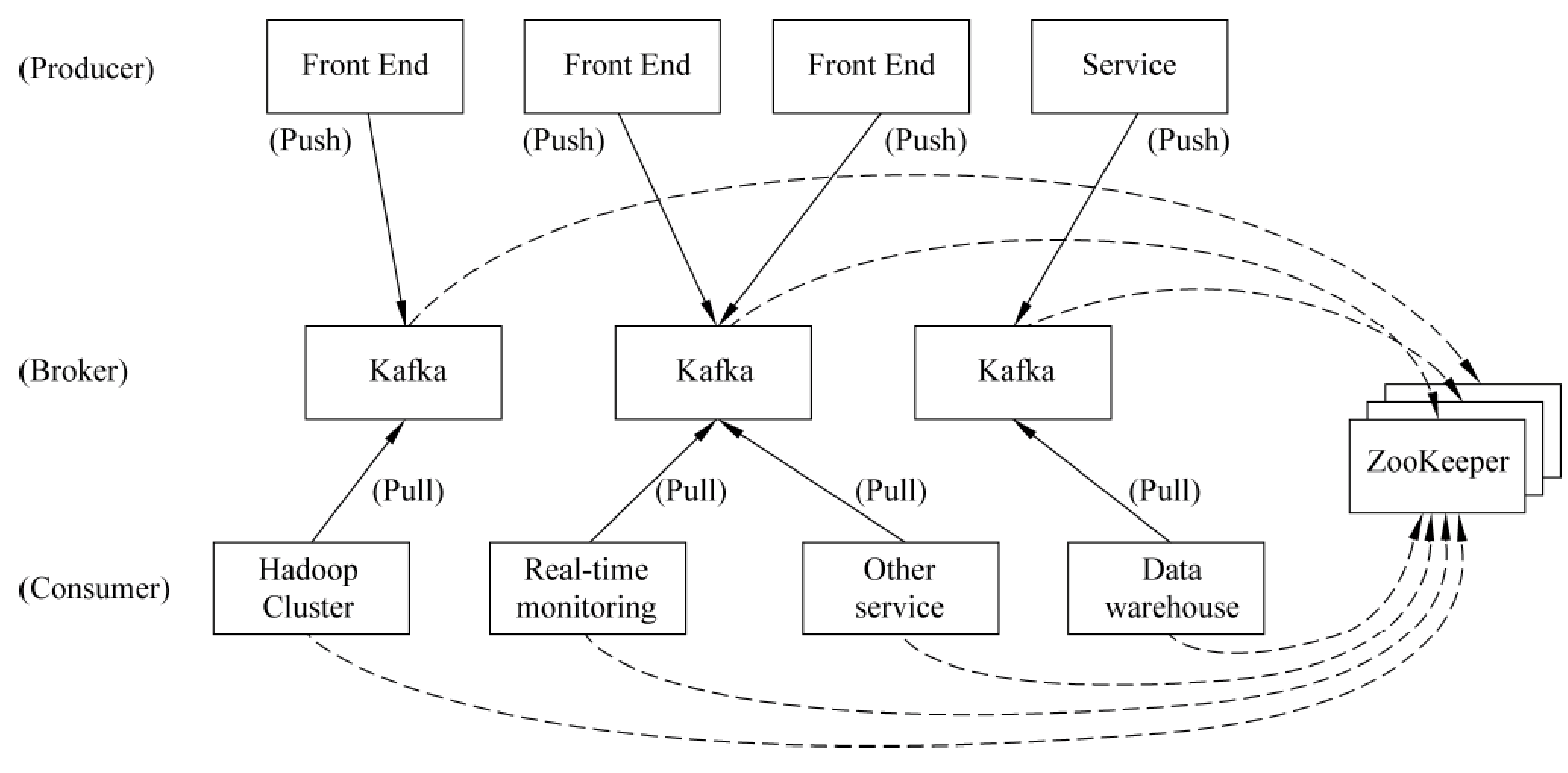


图 2.3 Kafka 的逻辑架构

从图 2.3 可以看出，Kafka 实际上是一个分布式的消息发布订阅系统，它主要有三种角色，分别为数据生产者(Producer)[②]，代理(Broker)[③] 和数据使用者(Consumer)[④]。Producer 向某个 Topic 发布(Push)消息，而 Consumer 订阅(Pull)某个 Topic 的消息，进而一旦有新的关于某个 Topic 的消息，Broker 会传递给订阅它的所有 Consumer。在 Kafka 中，消息是按 Topic 组织的，而每个 Topic 又会分为多个 Partition，这样便于管理

---

① 活动流是所有站点在对其网站使用情况做报表时要用到的数据中最常规的部分。活动流数据包括页面访问量(Page View)、被查看内容方面的信息以及搜索情况等内容。这种数据通常的处理方式是先把各种活动以日志的形式写入某种文件，然后周期性地对这些文件进行统计分析。运营数据指的是服务器的性能数据(CPU、I/O 使用率、请求时间、服务日志等等数据)。运营数据的统计方法种类繁多。

② 数据生产者 Producer 的任务是向 Broker 发送数据。Kafka 提供了两种 Producer 接口，一种是 LowLevel 接口，使用该接口会向特定的 Broker 的某个 Topic 下的某个 Partition 发送数据；另一种是 High Level 接口，该接口支持同步/异步发送数据，基于 ZooKeeper 的 Broker 自动识别和负载均衡。

③ 代理 Broker 采取了多种策略提高数据处理效率，包括 Sendfile 和 Zero Copy 等技术。

④ 数据使用者 Consumer 的作用是将日志信息加载到中央存储系统上。Kafka 提供了两种 Consumer 接口，一种是 Low Level 的，它维护到某一个 Broker 的连接，并且这个连接是无状态的，即每次从 Broker 上 Pull 数据时，都要告诉 Broker 数据的偏移量；另一种是 High Level 接口，它隐藏了 Broker 的细节，允许 Consumer 从 Broker 上 Push 数据而不必关心网络拓扑结构。

数据和进行负载均衡。同时,Producer 和 Broker 之间利用 ZooKeeper 进行了负载均衡,所有 Broker 和 Consumer 都会在 ZooKeeper 中进行注册,且 ZooKeeper 会保存一些元数据信息,如果某个 Broker 和 Consumer 发生了变化,其他所有的 Broker 和 Consumer 都会得到通知。

2.1.4 Time Tunnel

Time Tunnel[24]是一个高效的、可靠的、可扩展的实时数据传输平台,它是基于发布\订阅的消息模型开发的,支持消息多用户订阅。目前,Time Tunnel 在阿里巴巴广泛地应用于日志收集、数据监控、广告反馈、量子统计、数据库同步等领域。Time Tunnel 的主要功能就是实时完成海量数据的交换,因此它的业务逻辑主要也就有两个：一个是发布数据,将数据发送到 Time Tunnel;另一个是订阅数据,从 Time Tunnel 读取自己所关心的数据。Time Tunnel 的逻辑架构如图 2.4 所示。

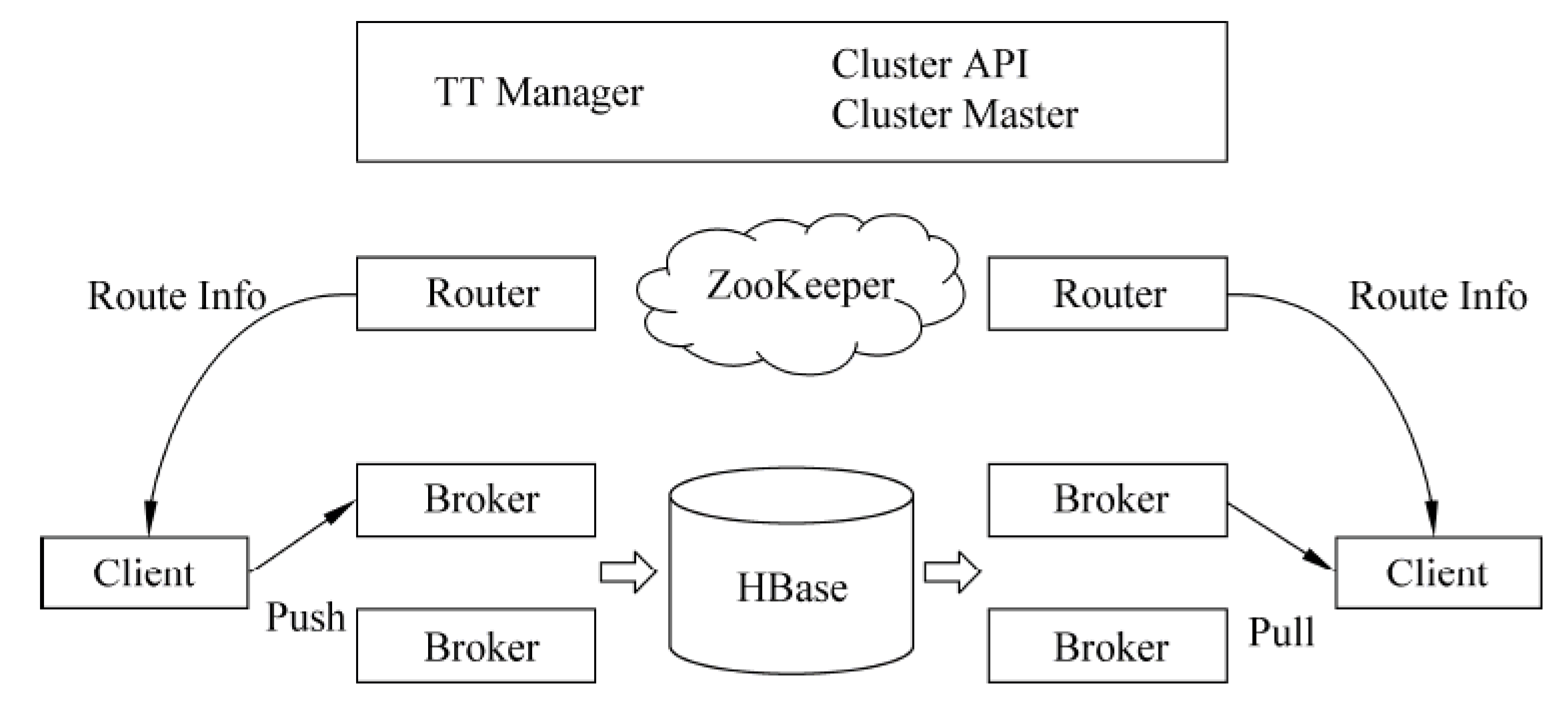


图 2.4 Time Tunnel 的逻辑架构

Time Tunnel 大概由 TT Manager、Client、Router、ZooKeeper 和 Broker 几部分组成。其中,TT Manager 负责对外提供队列申请、删除、查询和集群的管理接口;对内故障发现,发起队列迁移。Client 是一组访问 Time Tunnel 的 API,主要由三部分组成：安全认证 API,发布 API 和订阅 API,目前 Client 支持 Java、Python 和 PHP 三种语言。Router 是访问 Time Tunnel 的门户,主要负责路由、安全认证和负载均衡,为客户端提供路由信息,找到为消息队列提供服务的 Broker。ZooKeeper 是 Hadoop 的开源项目,其主要功能是状态同步,Broker 和 Client 的状态都存储在这里。Broker 是 Time Tunnel 的核心,负责消息的存储转发,承担实际的流量,进行消息队列的读写操作。

Client 访问 Time Tunnel 的第一步是向 Router 进行安全认证,如果认证通过,Router 根据 Client 要发布或者订阅的 Topic 对 Client 进行路由,使 Client 和正确的 Broker 建立连接,路由的过程包含负载均衡策略,Router 保证让所有的 Broker 平均地接收 Client 访问。Time Tunnel 作为一个实时数据传输平台具有以下特点。

1. 高效性

2KB 大小的消息,峰值每秒 4WTPS(每秒钟系统能够处理的交易或事务的数量)的

访问。淘宝三台服务器，每天处理 2.3TB（压缩后）数据，峰值每秒 50MB 流入流量、130MB 流出流量。

2. 实时性

90%的消息 5ms 以内送达。

3. 顺序性

如果开启了顺序传输功能，当没有故障发生时，Time Tunnel 保证消息的发布顺序和订阅顺序是一致的。

4. 可靠性

在数据存储方面，设计了内存→磁盘→Hadoop dfs 三级缓存机制，确保数据可靠；在系统方面，将服务器节点组织成环，在环里面每一个节点的后续节点是当前节点的备份节点，当某节点故障时，后续节点自动接管故障节点数据，以保证数据可靠性。

5. 可用性

单个节点故障，不影响系统正常运行。

6. 可扩展性

可以对系统进行横向和纵向扩展，横向扩展可以向现有的服务环里面增加节点，纵向扩展可以增加服务环。

2.1.5　Chukwa

Chukwa[25]是一个开源的用于监控大型分布式系统的数据收集系统，是构建在 Hadoop 的 HDFS 和 Map/Reduce 框架之上的，继承了 Hadoop 的可伸缩性和鲁棒性。Chukwa 还包含一个强大和灵活的工具集，可用于展示、监控和分析已收集的数据。Chukwa 的逻辑架构如图 2.5 所示。

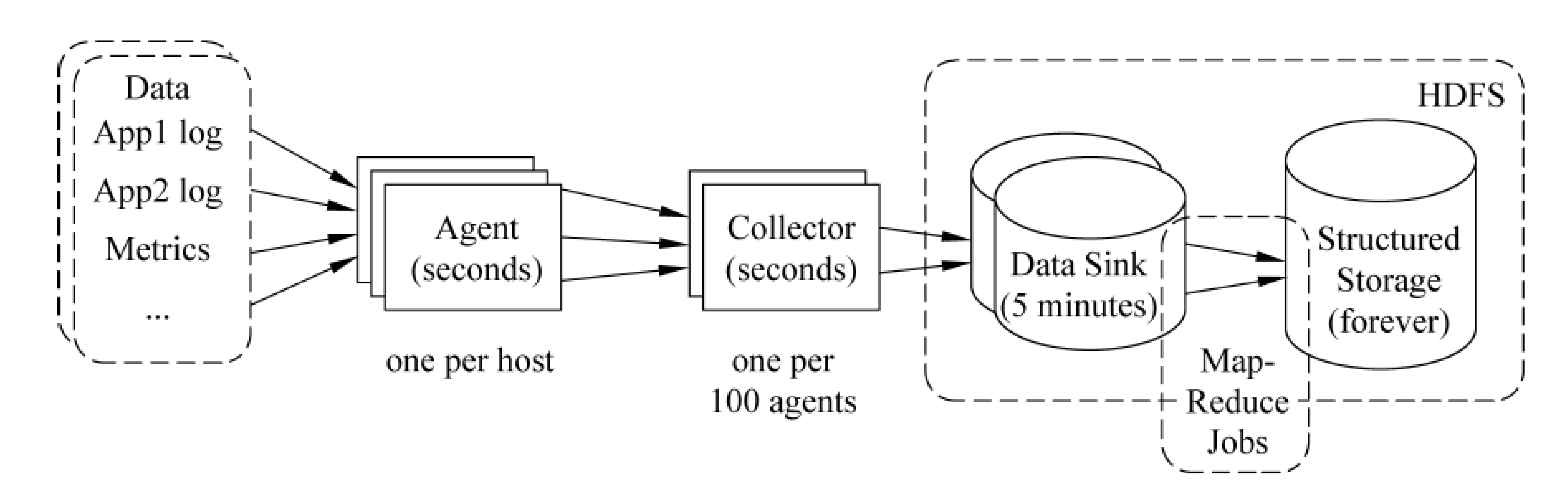


图 2.5　Chukwa 的逻辑架构

（图片来源：http://chukwa.apache.org/docs/r0.5.0/design.html）

从图 2.5 中可以看出，Chukwa 采用了 HDFS 作为存储系统，主要有三种角色，分别为：Adaptor，Agent，Collector。其中，Agent 主要负责给 Adaptor 提供各种服务，如启动和关闭 Adaptor，将数据通过 HTTP 传递给 Collector，定期记录 Adaptor 状态，以便 Crash 后恢复；Adaptor 是直接采集数据的接口和工具，一个 Agent 可以管理多个 Adaptor 的数据采集；Collector 主要负责收集 Agent 发送来的数据，并定时写入集群中；Map/Reduce Jobs 定时启动，负责把集群中的数据分类、排序、去重和合并。

Chukwa 可以用于监控大规模(2000＋以上的节点，每天产生数据量在 TB 级别) Hadoop 集群的整体运行情况并对它们的日志进行分析。对于集群的用户而言：Chukwa 展示他们的作业已经运行了多久，占用了多少资源，还有多少资源可用，一个作业为什么失败了，一个读写操作在哪个节点出了问题；对于集群的运维而言：Chukwa 展示了集群中的硬件错误，集群的性能变化，集群的资源瓶颈在哪里；对于集群的管理者而言：Chukwa 展示了集群的资源消耗情况，集群的整体作业执行情况，可以用以辅助预算和集群资源协调；对于集群的开发者而言：Chukwa 展示了集群中主要的性能瓶颈，经常出现的错误，从而可以着力重点解决重要问题。可以看出，Chukwa 从数据的产生、收集、存储、分析到展示的整个生命周期都提供了全面的支持。

2.2 大数据存储与管理技术

在大数据时代，随着数据规模的急速增长，数据类型复杂多样，其中主要以半结构化和非结构化为主，而传统的关系型数据库系统(如 Oracle、SQL Server 等)只能满足关系型数据的存储需求，无法满足半结构化和非结构化数据的存储需求，因此传统的数据存储和管理带来了一系列的挑战。例如，面对海量数据源源不断地产生，现有的单节点或共享磁盘架构面对海量数据存储的挑战；现有的基于结构化数据为主体的存储方案面对兼容无模式的非结构化数据的挑战；对采集到的海量、异构和混杂的大数据面对高效准确地传输、存储的挑战等。因此，大数据的存储与管理技术主要是解决复杂结构化、半结构化和非结构化大数据的存储与管理技术，并为其提供可扩展性强、可靠性高、性能卓越的数据存储、访问及管理解决方案，如开发可靠的分布式文件系统、分布式关系型数据库以及分布式非关系型数据库等解决方案。

针对大数据的数据量大的特点，可以采用分而治之的思想，即构建分布式存储系统。随着数据量的快速增加，分布式存储可以方便地增加存储节点，从而保证有足够的容量来存储数据，并使数据分布保持平衡状态。针对大数据的结构复杂多样的特点，可以根据每种数据的存储特点选择最合适的解决方案，如对非结构化数据采用分布式文件系统进行存储，对结构松散无模式的半结构化数据采用表存储、键值存储或面向文档的存储，对海量的结构化数据采用无共享的分布式并行数据库存储。依据数据结构特点所建议的存储方案见表 2.1。

表 2.1　依据数据结构特点所建议的存储方案

	结构化数据	非结构化数据	半结构化数据
定义	有数据结构描述信息的数据	不方便用固定结构来表现的数据	介于完全结构化数据和完全非结构化数据之间的数据
特点	先有结构，再有数据	只有数据，没有结构	先有数据，再有结构
存储	分布式关系型数据库	分布式文件系统	分布式非关系型数据库

221　分布式文件系统

大数据存储与管理面临的首要问题就是如何构建一个大型的分布式文件系统①，从而为海量数据进行有效处理和分析提供底层存储支撑。采用分布式系统来存储海量数据时，应该考虑以下三个核心的需求[43,44]。

(1) Consistency(一致性)：在分布式系统中的所有数据备份，在同一时刻是否是同样的值，等同于所有节点访问同一份最新的数据副本。

(2) Availability(可用性)：在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求，即对数据更新要具备高可用性。

(3) Partition Tolerance(分区容错性)：以实际效果而言，分区相当于对通信的时限要求，系统如果不能在时限内达成数据一致性，就意味着发生了分区的情况，必须就当前操作在 Consistency(一致性)和 Availability(可用性)之间做出选择。

针对分布式系统设计和部署的三个核心需求，最初是由 Brewer 在 2000 年的 PODC (Principles Of Distributed Computing)会议上提出的著名的 CAP 理论[43]。2002 年，Seth Gilbert 和 Nancy Lynch 证明了这一理论[44]，使之成为一个定理。根据定理，分布式系统只能满足三项中的两项而不可能满足全部三项。而对于分布式数据系统而言，分区容错性是基本要求，否则就不称其为分布式系统了。因此，架构设计师不要把精力浪费在设计如何能同时满足三者的完美分布式系统上，而是应该进行权衡取舍。这也意味着分布式系统的设计过程，也就是根据业务特点在 Consistency(一致性)和 Availability(可用性)之间寻求平衡的过程，要求架构师真正理解系统需求，把握业务特点。

目前典型的分布式文件系统有 Lustre[28-30]、GFS[31] (Google File System)、GlusterFS、PVFS(Parallel Virtual File System)、FastDFS、NFS、MogileFS、FreeNAS、OpenAFS、MooseFS、QFS(Quantcast File System)、Ceph、HDFS(Hadoop Distributed File System)等。它们的许多设计理念类似，同时也各有自己的特色。下面对其中的几款分布式文件系统进行简单介绍。

1. Lustre

Lustre[28-30]文件系统是一个开源的、基于对象存储技术的集群并行文件系统，是一个

① 分布式文件系统(Distributed File System，DFS)是指网络中的多个存储节点通过网络组织起来，文件系统管理的物理存储资源不一定直接连接在本地节点上，而是通过计算机网络与节点相连。

大规模的、安全可靠的，具备高可用性的集群文件系统，它是由 Sun 公司开发和维护的。该项目主要的目的就是开发下一代的集群文件系统，能够支持数万客户端系统、PB 级存储容量、数百 GB 的聚合 I/O 吞吐量，使其具有完美的可伸缩性、安全性和可管理性等。Lustre 是 Scale-Out 存储架构，借助强大的横向扩展能力，通过增加服务器即可方便扩展系统总存储容量和性能，非常适合众多客户端并发进行大文件读写的场合，但目前对于小文件应用非常不适用，尤其是海量小文件应用(Lots Of Small Files，LOSF)。

Lustre 集群组件包含 MDS(The Metadata Server，元数据服务器)、MDT(Metadata Target，元数据存储节点)、OSS(An Object Storage Server，对象存储服务器)、OST(An Object Storage Target，对象存储节点)、Client(Lustre Clients，客户端)，以及连接这些组件的高速网络(High Speed Interconnect)。比较合理的 Lustre 文件系统集群结构如图 2.6 所示。

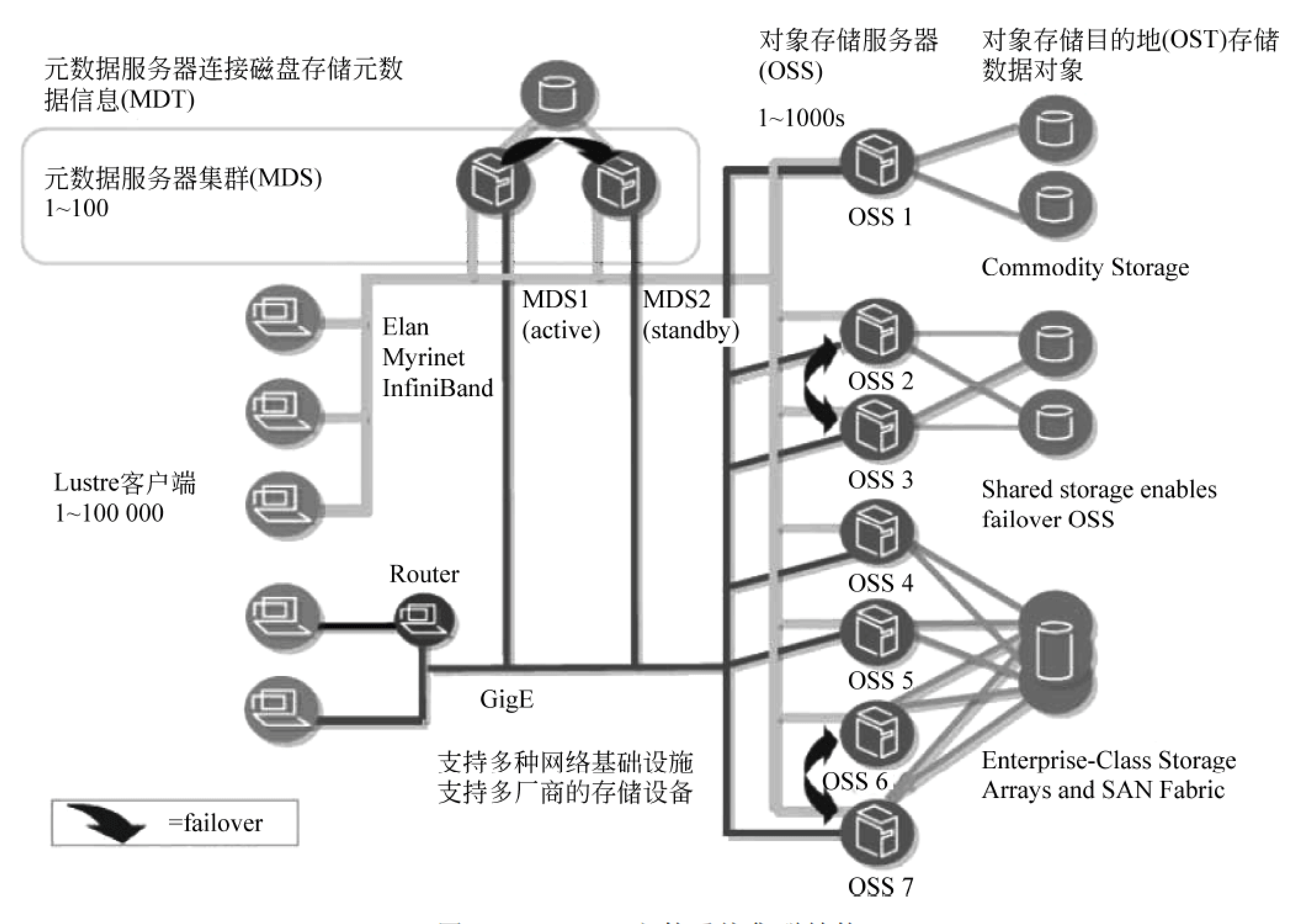


图 2.6 Lustre 文件系统集群结构

MDS 负责管理元数据，提供一个全局的命名空间，Client 可以通过 MDS 读取到保存于 MDT 之上的元数据。在 Lustre 中 MDS 可以有两个，采用了 Active-Standby 的容错机制，当其中一个 MDS 不能正常工作时，另外一个后备 MDS 可以启动服务。MDT 只能有一个，不同 MDS 之间共享访问同一个 MDT。

MDT 为单独的文件系统的元数据信息提供了后端存储区。MDT 存储了 MDS 上元数据的文件名、目录、权限和文件布局。一个文件系统只能有一个 MDT，不同的 MDS 之间共享同一个 MDT。

OSS 负载提供 I/O 服务，接受并服务来自网络的请求。通过 OSS，可以访问到保存在 OST 上的文件数据。一个 OSS 对应 2～8 个 OST，其存储空间可以高达 8TB。OST 上的文件数据是以分条的形式保存的，文件的分条可以在一个 OSS 之中，也可以保存在多个 OSS 中。

OST 负责实际数据的存储，处理所有客户端和物理存储之间的交互。这种存储是基于对象（Object-based）的，OST 将所有的对象数据放到物理存储设备上，并完成对每个对象的管理。OST 和实际的物理存储设备之间通过设备驱动程序来实现交互。通过驱动程序的作用，Lustre 可以继承新的物理存储技术以及文件系统，实现对物理存储设备的扩展。存储在 OST 上的文件可以是普通文件，也可以是复制文件。

Client 是通过标准的 POSIX 接口向用户提供对文件系统的访问，主要负责同 OST 进行文件数据的交互，包括文件数据的读写、对象属性的改变等；同 MDS 进行元数据的交互，包括目录管理、命名空间管理等。

目前 Lustre 文件系统最多可以支持 100 000 个 Client，1000 个 OSS 和两个 MDS 节点。Lustre 系统中可以同时运行 1～3 个功能模块。不过 Lustre 一般运行于高性能计算机系统之上，为了提高 Lustre 文件系统的性能，通常 MDS、OSS 和 Client 是分开运行在 Lustre 不同的节点之上的。实验与应用已经证明，Lustre 文件系统的性能和可扩展性都不错；还拥有基于对象的智能化存储、安全的认证机制、比较完善的容错机制等优点。值得注意的是，Lustre 还实现了部分文件锁；为了满足高性能计算系统的需要，Lustre 针对大文件的读写进行了优化，为集群系统提供较高的 I/O 吞吐率，是解决目前存储 I/O 瓶颈较好的方式；而且 Lustre 在可用性和扩展性方面的性能优越。然而，Lustre 需要特殊设备的支持，并且 Lustre 目前还没实现 MDS 的集群管理，今后很有可能成为 Lustre 系统中的瓶颈。

2. GFS

Google 文件系统（Google File System，GFS）[31]是 Google 自己研发的一个适用于大规模分布式数据处理相关应用的、可扩展的分布式文件系统，它运行于廉价的普通硬件上，提供容错功能，在保证系统可靠性和可用性的同时，减少了系统的成本。GFS 的系统架构如图 2.7 所示。

GFS 将整个系统的节点分为三类角色：Client（客户端）、Master（主服务器）和 Chunk Server（数据块服务器）。Client 是 GFS 提供给应用程序的访问接口，它是一组专用接口，不遵守 POSIX 规范，以库文件的形式提供。应用程序直接调用这些库函数，并与该库链接在一起。Master 是 GFS 的管理节点，主要维护系统的元数据，包括文件及 Chunk 名字空间，GFS 文件到 Chunk 之间的映射，Chunk 位置信息。它也负责整个系统的全局控制，如 Chunk 租约管理，垃圾回收无用 Chunk，Chunk 复制等。Chunk Server 负责具体的存储工作。数据以文件的形式存储在 Chunk Server 上，Chunk Server 的个数可以有多个，它的数目直接决定了 GFS 的规模。GFS 将文件按照固定大小进行分块，默认是 64MB，每一块称为一个 Chunk（数据块），每个 Chunk 都有一个对应的索引号（Index）。

客户端在访问 GFS 时，首先访问 Master 节点，获取将要与之进行交互的 Chunk

Server 信息，然后直接访问这些 Chunk Server 完成数据存取。GFS 的这种设计方法实现了控制流和数据流的分离。Client 与 Master 之间只有控制流，而无数据流，这样就极大地降低了 Master 的负载，使之不成为系统性能的一个瓶颈。Client 与 Chunk Server 之间直接传输数据流，同时由于文件被分成多个 Chunk 进行分布式存储，Client 可以同时访问多个 Chunk Server，从而使得整个系统的 I/O 高度并行，系统整体性能得到提高。

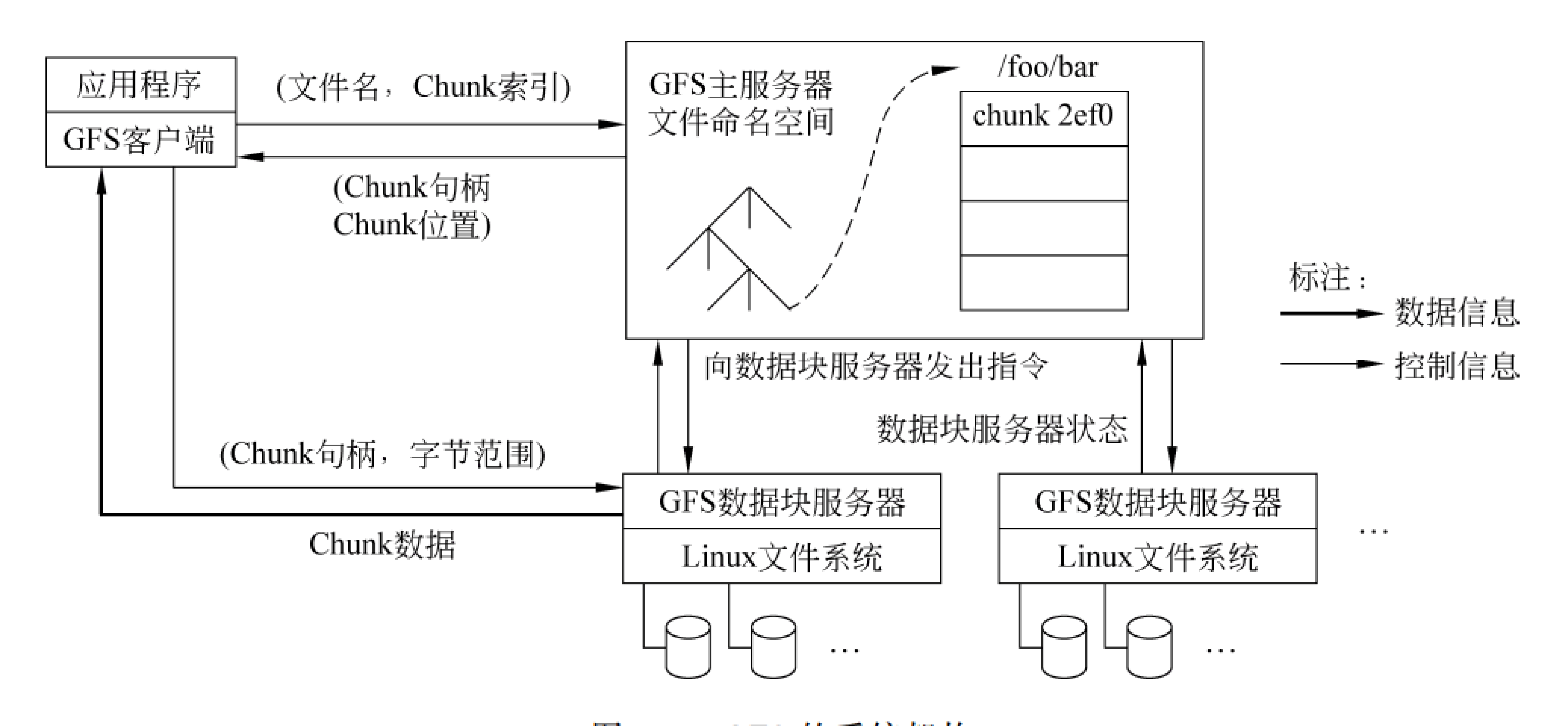


图 2.7　GFS 的系统架构

（图片来源：来自论文 *The Google File System*）

3. PVFS

PVFS[32](Parallel Virtual File System)项目是 Clemson 大学为了运行 Linux 集群而创建的一个并行虚拟文件系统。PVFS 是基于传统的 C/S 架构进行设计，整个文件系统由管理节点、计算节点和 I/O 节点三大部分组成。管理节点负责处理文件的元数据，计算节点用来执行各种计算任务，I/O 节点则主要负责数据文件的存储和读写，并负责给计算节点提供所需的数据。PVFS 系统架构如图 2.8 所示。

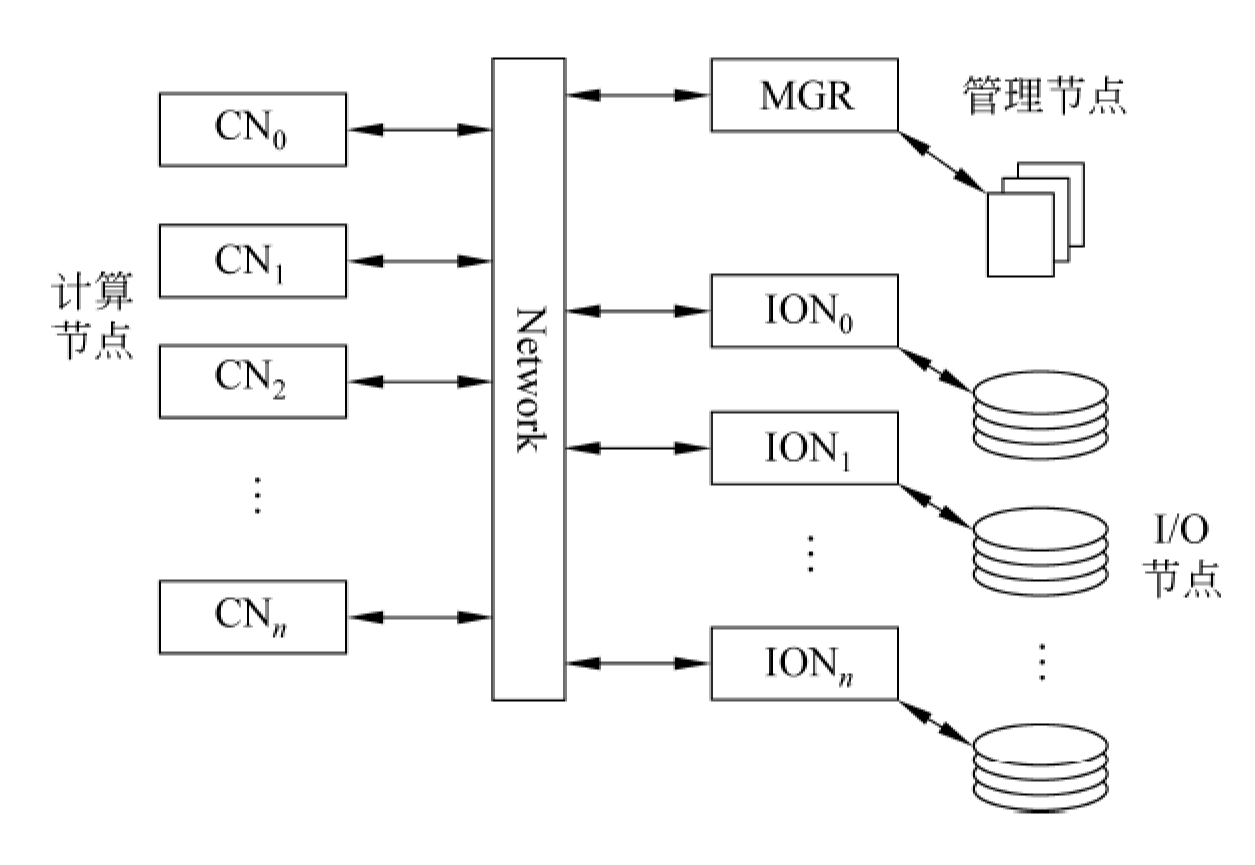


图 2.8　PVFS 的系统架构

（图片来源：http://www.pvfs.org/documentation/）

PVFS 系统中有且只有一个管理节点，一个或者多个计算节点和 I/O 节点。在整个集群系统范围内，PVFS 使用一致的全局命名空间。另外，PVFS 应用对象存储的概念，将数据文件条块化为多个对象并分别存储到多个存储节点上。由于在网络通信方面，PVFS 只支持 TCP 网络通信协议，这使得其灵活性不足。此外，由于 PVFS 应用对象存储的概念进行数据文件的存储，其在处理小文件时性能也不太理想。PVFS 集中的元数据管理成为整个系统的瓶颈，可扩展性受到一定限制，而且数据没有采取相应的容错机制，不具备动态扩展功能，系统的可用性有待提高。

4. HDFS

HDFS 是 Hadoop Distributed File System 的缩写，意为 Hadoop 分布式文件系统。它被设计成运行在普通商用硬件上并具有高容错率的文件系统，为应用程序数据提供了很高的吞吐量，适合处理大量数据的应用程序。HDFS 最初是作为 Apache 的 Nutch 网站搜索引擎项目的基础设施而建立的，现在演变为 Apache 的 HDFS 项目，有关 HDFS 的相关内容请查看第 4 章中“Hadoop 分布式文件系统”相关内容。

222 分布式数据库

分布式数据库是一个数据集合，这些数据在逻辑上属于同一个系统，但物理上却分散在计算机网络的若干站点上，并且要求网络的每个站点具有自治的处理能力，能执行本地的应用。因此，分布式数据库的两个重要特点：分布性和逻辑相关性。依据存储的数据结构不同，分布式数据库可分为分布式关系型数据库和分布式非关系型数据库。

1. 关系型数据库

分布式关系型数据库是一种强调遵循 ACID 原则[46]的数据存储系统，即数据库事务正确执行应用遵循以下 4 个基本要素。

1） Atomicity（原子性）

事务的操作要么全部执行，要么全部不执行，不会结束在中间某个环节，从而保证数据库一致性状态。

2） Consistency（一致性）

事务的正确性，串行性，并发执行的多个事务，其操作的结果应与以某种顺序串行执行这几个事务所得的结果相同。

3） Isolation（隔离性）

虽然可以有多个事务同时执行，但是单个事务的执行不应该感知其他事务的存在，因此事务执行的中间结果应该对其他并发事务隐藏。

4） Durability（持久性）

当事务提交后，其操作的结果将永久化，而与提交后发生的故障无关。

为了保证数据库的 ACID 特性，必须尽量按照其要求的范式进行设计，关系型数据库中的表都是存储一个格式化的数据结构。每个元组字段的组成都是一样的，即使不是每

个元组都需要所有的字段，但数据库会为每个元组分配所有的字段，这样的结构可以便于表之间进行链接等操作，但从另一个角度来说它也是关系型数据库性能瓶颈的一个因素。一些常见的关系型数据库系统产品，比如 Federated PostgreSQL、Oracle、Federated MySQL、Oracle Exadata、Ingres、Sybase、Informix、Greenplum 和 IBM DB2 等，大都提供了对分布式数据库的不同程度的支持。下面对其中的几款分布式关系型数据库进行简单介绍。

(1) Greenplum[47]是基于 Hadoop 的一款分布式关系型数据库产品，可以通过标准的 SQL 对 Greenplum 中的数据进行访问存取等操作，在处理海量数据方面相比传统数据库有着较大的优势。该数据库实际上是由数个独立的数据库服务组合成的逻辑数据库，即关系型数据库集群。这种数据库集群采用的是 MPP(Massively Parallel Processing)架构，如图 2.9 所示。

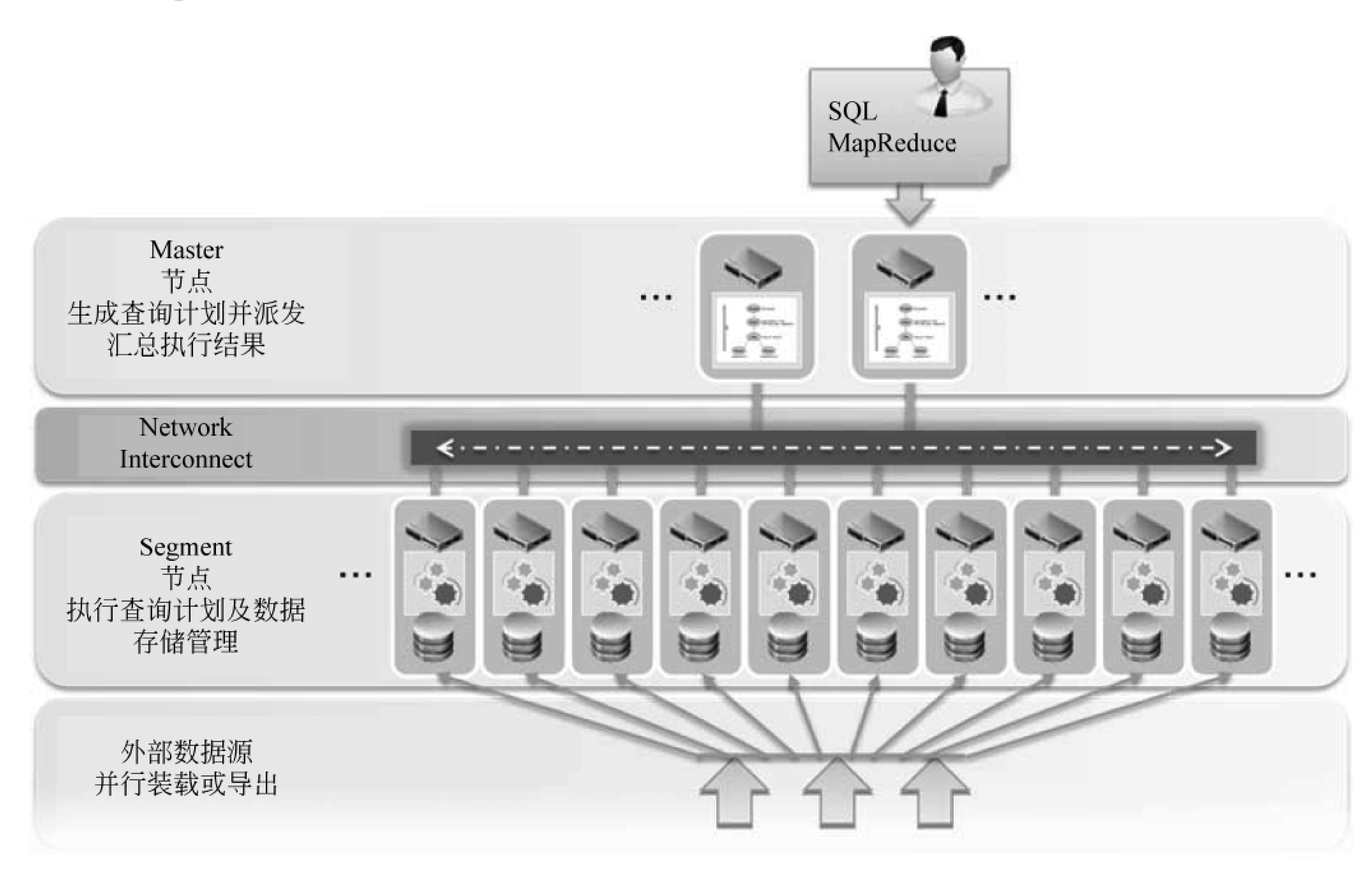


图 2.9　Greenplum 体系架构

(图片来源：http://www.greenplum.com/)

Greenplum 数据库由 Master 节点和 Segment 节点通过 Interconnect 互连组成，其中各节点的功能如下。

Master 节点主要负责：建立与客户端的连接和管理；SQL 的解析并形成执行计划；执行计划向 Segment 的分发；收集 Segment 的执行结果；Master 不存储业务数据，只存储数据字典。

Segment 节点主要负责：业务数据的存储和存取；用户查询 SQL 的执行。

Network Interconnect 主要负责：基于超级计算的软件 Switch 内部连接层；基于通用的网卡和交换机。



从各节点的功能可以看出，Master 节点只负责应用的连接，生成并拆分执行计划，把执行计划分配给 Segment 节点，以及返回最终结果给应用，它只存储一些数据库的元数据，不负责运算，因此不会成为系统性能的瓶颈。Segment 节点存储用户的业务数据，并根据得到的执行计划，负责处理业务数据，将业务数据平均分布到每个 Segment 节点。当进行数据访问时，首先所有 Segment 节点并行处理与自己有关的数据，Segment 节点之间通过 Network Interconnect 进行数据交互。Segment 节点越多，数据处理速度就越快。

(2) Oracle Exadata 是 Oracle 公司研发的一款 Oracle 数据库一体机，专为运行 Oracle 数据库而构建。Oracle Exadata 在设计时，借鉴了 Hadoop HDFS 等分布式计算集群的设计理念，使 Oracle 数据库的运行环境突破了传统的基于共享存储的 Scale-up 架构[①]，而使用了基于 Scale-out[②] 的智能存储节点的架构，从而极大地提高了 Oracle 数据库在处理海量数据方面的性能，该系统的简化体系架构如图 2.10 所示。

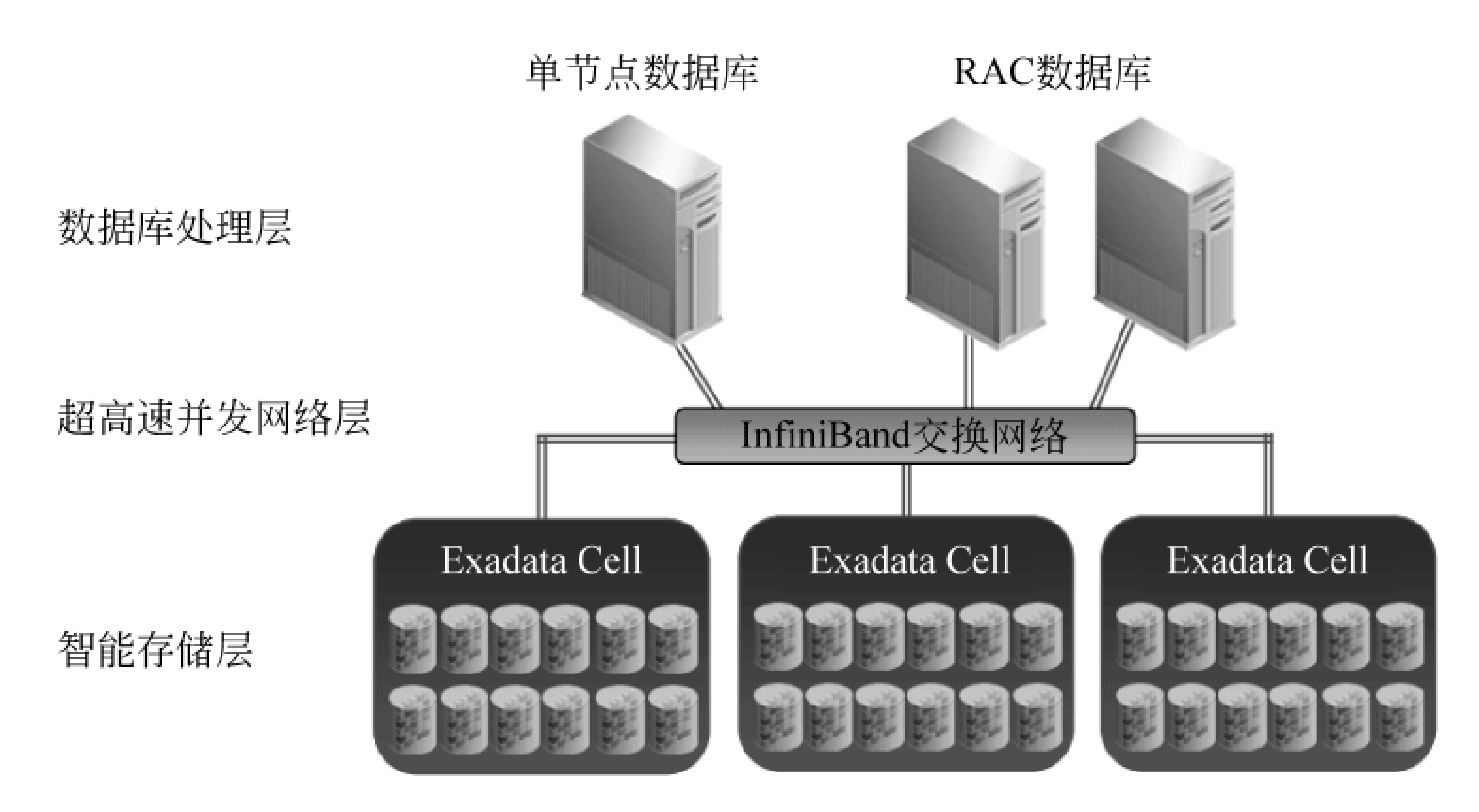


图 2.10　Oracle Exadata 体系架构

（图片来源：http://www.oracle.com/）

如图 2.10 所示，Oracle Exadata 是由数据库处理层、超高速并发网络层和智能存储层构成。图中数据库处理层中的数据库一个是数据库服务器部署的 Real Application Clusters(RAC)数据库(可以用于生产数据库)，另一个是单实例数据库(可以用于测试和开发)；超高速并发网络层是采用 InfiniBand(高带宽，低延迟)交换网络，主要功能是实现数据库处理层和智能存储层之间的互连，同时还为 Oracle 数据库 RAC 节点提供高性能集群互连；智能存储层可采用创新的基于 Scale-out[②] 的智能存储节点，不仅提供了高性能和高可扩展性，同时还采用了智能闪存缓存等技术提高了海量数据的存储和处理性能。

---

① Scale-up 架构，即纵向扩展架构，是利用现有的存储系统，通过不断增加存储容量来满足数据增长的需求。该架构的优点在于不必增加基础设施设备(如：扩展网络连接能力)就可完成容量的升级，即只有容量升级的成本，不会增加控制器或基础设施的开销；缺点在于在容量增长的同时，需要增加物理空间、电力消耗和散热能力等。

② Scale-out 架构，即横向扩展架构，是以节点为单位，通过增加具有完整功能的节点进行扩展。该架构的优点在于一个 Scale-out 系统可以有很多节点，而且节点之间的内部物理互连距离可以很远，并且容量和性能会同步增长；缺点在于系统横向扩展时不仅需要电力、散热和机房空间，而且还需要增加容量、控制部件和基础设施成本。

2. 非关系型数据库

分布式非关系型数据库是指那些非关系型的、分布式的、不保证遵循 ACID 原则的数据存储系统。非关系型(NoSQL)分布式数据库的理论基础一般遵循 BASE 模型[48],该模型是基于 CAP 理论逐步演化而来,是 Basically Available(基本可用)、Soft-state(软状态/柔性事务)和 Eventually Consistent(最终一致性)三个词组的简写。BASE 模型是不遵循 ACID 原则的,它完全不同于 ACID 模型,通过牺牲强一致性,来获得基本可用性和柔性可靠性并要求达到最终一致性。

注:最终一致性(Eventually Consistent)是指:经过一段时间以后,更新的数据会到达系统中的所有相关节点。

分布式非关系型数据库的数据存储不需要固定的表结构,通常也不存在连接操作。在海量的数据存取上具备关系型数据库无法比拟的性能优势。非关系型数据库可按功能分为文档数据库(Document Databases)、图数据库(Graph Databases)、键值数据库(Key/Value Databases)、列存储数据库(Columnar Databases)和内存数据网络(In-Memory Data Grids)。下面针对不同类型的分布式非关系型数据库产品进行简单举例介绍。

1) 文档数据库

MongoDB:是一个可扩展的、高性能、开源、模式自由的文档型数据库。

CounchDB:是一个使用 JSON 作为数据格式的文档数据库,并可以通过视图来操纵文档的组织和呈现。

Couchbase:是一个分布式的面向文档的 NoSQL 数据库管理系统,该系统联合了 CouchDB 的简单和可靠以及 Memcached 的高性能以及 Membase 的伸缩性。

RavenDB:是一个新的.NET、支持 Linq 的开源文档数据库,旨在 Windows 平台下提供一个高性能,结构简单、灵活,可扩展的 NoSQL 存储。

2) 图数据库

Neo4j:是一个高性能的 NoSQL 图形数据库,使用图相关的概念来描述数据模型,把数据保存为图中的节点以及节点之间的关系,并且很多应用中数据之间的关系,可以很直接地使用图中节点和关系的概念来建模。

InfiniteGraph:是一个图数据库,用来维持和遍历对象间的关系,支持分布式数据存储。

AllegroGraph:是一个基于 W3C 标准的为资源描述框架构建的图形数据库,它为处理链接数据和 Web 语义而设计,支持 SPARQL、RDFS++ 和 Prolog。

3) 键值数据库

Riak:是一个开源分布式键值数据库,支持数据复制和容错。

Redis:是一个开源的键值存储数据库,支持主从式复制、事务,Pub/Sub、Lua 脚本,还支持给 Key 添加时限。

Dynamo:是一个键值分布式数据存储,它直接由亚马逊 Dynamo 数据库实现,在亚马逊 S3 产品中使用。

Oracle NoSQL Database:来自 Oracle 的键值 NoSQL 数据库,它支持事务 ACID 和

JSON，具备数据备份和分布式键值存储系统。

Voldemort：具备数据备份和分布式键值存储系统。

Aerospike：是一个键值存储，支持混合内存架构，通过强一致性和可调一致性保证数据的完整性。

4）列存储数据库

Cassandra：是列存储数据库，支持跨数据中心的数据复制，它的数据模型提供列索引，log-structured 修改，支持反规范化，实体化视图和嵌入超高速缓存。

HBase：是一个开源、分布式、面向列存储的模型，在 Hadoop 和 HDFS 之上提供了像 Bigtable 一样的功能。

Amazon SimpleDB：是一个非关系型数据存储，开发者可使用 Web 服务请求存储和查询数据项。

Apache Accumulo：是一个有序的、分布式键值数据存储，基于 Google 的 Bigtable 设计，建立在 Apache Hadoop、ZooKeeper 和 Thrift 技术之上。

Hypertable：是一个开源、可扩展的数据库，模仿 Bigtable，支持分片。

5）内存数据网络

Hazelcast：是一个开源数据分布平台，它允许开发者在数据库集群之上共享和分割数据。

Oracle Coherence：提供了常用数据的快速访问能力，一致性支持事务处理能力和数据的动态划分。

Terracotta BigMemory：包括一个 Ehcache 界面、Terracotta 管理控制台和 BigMemory-Hadoop 连接器。

GemFire：是一个分布式数据管理平台，也是一个分布式的数据网格平台，支持内存数据管理、复制、划分、数据识别路由和连续查询。

Infinispan：是一个基于 Java 的开源键值 NoSQL 数据存储和分布式数据节点平台，支持事务，Peer-to-Peer 及 Client/Server 架构。

GridGain：是一个分布式、面向对象、基于内存、SQL＋NoSQL 键值数据库，支持 ACID 事务。

2.3　大数据分析与挖掘技术

大数据分析与挖掘技术是大数据处理生命周期中的核心技术，因为大数据的价值产生于分析过程，即要从海量的、不完整的、有噪声的、模糊的、绝对随机的数据中发现隐含在其中有价值的、潜在有用的知识，从而获得洞察。大数据的分析与挖掘技术主要涉及传统的数据分析与挖掘方法、大数据分析与挖掘方法、大数据分析与挖掘框架等。

2.3.1　传统数据分析与挖掘方法

传统数据分析与挖掘方法主要是针对结构化数据和事务处理的关系型数据库为主，根据不同应用的需求在此基础上构建数据仓库，并选择相关数据进行分析，常用的分析与

挖掘方法有数据挖掘、机器学习、统计分析等。这些传统的数据分析与挖掘方法在处理相对较少的结构化数据时比较有效，在面对大数据分析与挖掘时，有些传统的数据分析与挖掘方法可以直接应用于大数据的分析与挖掘，有的数据分析与挖掘方法则需要做出相应的调整。下面对适合大数据技术的传统数据分析与挖掘方法进行简单举例分析。

1. 分类分析

分类分析是数据挖掘、机器学习和模式识别中一个重要的研究领域，通过对已知类别训练集的分析，从中发现分类规则，从而来预测新数据的类型。分类分析在大数据中的应用非常频繁，而常见的分类算法也有很多，不同分类算法又有很多不同的变种，常用的大数据分类算法有：逻辑回归(Logistic Regression)、贝叶斯(Bayesian)、支持向量机(Support Vector Machines)、感知器(Perceptron and Winnow)、神经网络(Neural Network)、随机森林(Random Forests)、有限玻耳兹曼机(Restricted Boltzmann Machines)等。

1）逻辑回归

逻辑回归是一种利用预测数值型或离散型变量来预测某种事件的可能性，是一个被广泛应用在实际场景中的算法。逻辑回归可以用来回归，也可以用来分类，主要是二分类，即将两个不同类别的样本分开。

2）贝叶斯

贝叶斯分类器的分类原理是通过某对象的先验概率，利用贝叶斯公式计算出其后验概率，即该对象属于某一类的概率，选择具有最大后验概率的类作为该对象所属的类。

3）支持向量机

支持向量机 SVM 是 AT&T Bell 实验室的 V. Vapnik 提出的针对分类和回归问题的统计学习理论。支持向量机 SVM 对于线性可分情况时，可直接进行分析；对于线性不可分的情况，通过使用非线性映射算法将低维输入空间线性不可分的样本转化为高维特征空间使其线性可分，从而使得高维特征空间采用线性算法对样本的非线性特征进行线性分析。

4）感知器

感知器是由美国计算机科学家罗森布拉特(F. Roseblatt)于 1957 年提出的，可谓是最早的人工神经网络，它具有一层神经元、采用阈值激活函数的前向网络，通过对网络权值的训练，可以对一组输入矢量的响应达到元素为 0 或 1 的目标输出，从而实现对输入矢量分类的目的。

5）神经网络

神经网络对外界的输入样本具有很强的识别能力，可以发现输入样本自身的联系和规律以及输入样本和期望输出之间的非线性规律。该分类算法的重点是构造阈值逻辑单元，一个值逻辑单元是一个对象，它可以输入一组加权系数的量，对它们进行求和，如果这个和达到或者超过了某个阈值，则输出一个量。

6）随机森林

随机森林是由 Breiman 提出的一种基于 CART 决策树的组合分类器，该算法的核心

思想是用随机的方式建立一个森林,森林里面有很多的决策树,随机森林中的任意两棵决策树是相对独立的。对于新来的测试样本,通过每棵决策树都对它进行分类决策,最后的分类结果由投票法得出,并且其输出的类别是由个别树输出的类别的众数而定。

7）有限玻耳兹曼机

有限玻耳兹曼机是 Hinton 和 Sejnowski 于 1986 年提出的一种可用随机神经网络来解释的概率图模型。该算法是在玻尔兹曼机的基础上提出的一类具有两层结构、对称连接且无自反馈的随机神经网络模型,层间全连接,层内无连接,其输出只有两种状态(未激活和激活状态),而状态的具体取值则根据概率统计法则来决定。

在应用这些分类算法时,需要根据特定的应用场景进行算法的选择,因为不同的分类算法在不同的数据集上表现的效果不同。分类算法的应用也非常广泛,如银行中风险评估、市场营销客户类别细分、文本检索、搜索引擎分类、安全领域中的入侵检测等。例如,腾讯广点通就是大数据分类算法的一个典型应用,它是基于腾讯大社交网络体系的效果广告营销产品,在腾讯大社交平台的海量用户积累的基础上,运用大数据技术,进行以人为核心的数据挖掘,实现精准的广告推荐。这种基于大数据的实时精准推荐系统的核心思想就是针对不同的推荐场景,采用不同的数据和不同的算法策略。

2. 聚类分析

聚类是在给定的数据集合中寻找同类的数据子集合,每一个子集合形成一个类簇,同类簇中的数据具有更大的相似性。聚类分析能在无先验信息的条件下,探测数据在特征空间中的分布或类别结构,从而提供潜在有价值的信息。由于大数据具有数据类型多、海量性等特点,因此针对大数据进行聚类分析无论从理论、算法还是实践方面仍有很多极具挑战性的问题亟待解决。常用的大数据聚类算法有 K 均值(K-means Clustering)、模糊 K 均值(Fuzzy K-means)、期望最大化聚类(Expectation Maximization)、均值漂移聚类(Mean Shift Clustering)、层次聚类(Hierarchical Clustering)、狄里克雷过程聚类(Dirichlet Process Clustering)、LDA 聚类(Latent Dirichlet Allocation)、Canopy 聚类(Canopy Clustering)、谱聚类(Spectral Clustering)等。下面简单介绍几种常用聚类算法。

1）K 均值

K 均值聚类算法是最为经典的基于划分的聚类方法,该算法的基本思想是以空间 K 个点为中心进行聚类,对最靠近它们的对象归类,通过迭代的方法,逐次更新各聚类中心的值,直至得到最好的聚类结果。

2）期望最大化聚类

期望最大化聚类算法是在概率模型中寻找参数最大似然估计或者最大后验估计的算法,其中概率模型依赖于无法观测的隐藏变量。因此,该算法适用于一种解决存在隐含变量优化问题的有效方法。

3）均值漂移聚类

均值漂移聚类算法最早是由 Fukunaga 等人于 1975 年在一篇关于概率密度梯度函数的估计中提出来的,其基本思想是先算出当前点的偏移均值,移动该点到其偏移均值,

然后以此为新的起点继续移动，直到满足一定的条件结束。

4）层次聚类

层次聚类算法是通过对数据集按照某种方法进行层次分解，直到满足某种条件为止。按照分类原理的不同，又可以细分为凝聚和分裂两种方法，凝聚的层次聚类是一种自底向上的策略，而分裂的层次聚类采用自顶向下的策略。

5）谱聚类

谱聚类是一种基于图论的聚类方法，它能够识别任意形状的样本空间且收敛于全局最优解，其基本思想是利用样本数据的相似矩阵进行特征分解后得到的特征向量进行聚类。该聚类算法可直接用于大数据集，并广泛应用于计算机视觉、语音识别、文本挖掘等领域。

3. 关联规则

关联规则概念最早是由 Agrawal 等人于 1993 年提出的，即发现大量数据中数据项之间的相关联系。关联规则比较经典的算法有 Apriori 算法和频繁树（FP-Tree）算法，后来又纷纷提出了各种改进算法或者不同的算法，下面就对这两种算法进行简单介绍。

1）Apriori 算法

Apriori 算法是一种挖掘关联规则的频繁项集算法，其核心思想是基于两阶段频集思想的递推算法。该关联规则在分类上属于单维、单层、布尔关联规则，主要用来在大型数据库上进行快速挖掘关联规则。

2）FP-Tree 算法

FP-Tree 算法是由 J. Han 等人于 2000 年提出的，针对 Apriori 算法的固有缺陷提出了不产生候选频繁项集的方法即 FP-Tree 算法。该算法直接将事务数据库压缩成一个频繁模式树，然后通过这棵树生成关联规则。

Apriori 算法和 FP-Tree 算法是基于单点的算法，无法满足对海量数据的处理需求。因此，研究人员提出了多种基于 Apriori 算法和 FP-Tree 算法的并行挖掘算法，其基本思想是将数据平均分配到 N 个计算节点上，在每个节点上都采用类似 Apriori 和 FP-Tree 的算法。这样解决了挖掘效率问题，但是由于并行计算是由很多计算节点组成，仍然会带来如节点失效、负载不均衡等很多问题。

4. 回归分析

回归分析（Regression Analysis）是确定两种或两种以上变量之间相互依赖的定量关系的一种统计分析方法，即通过规定因变量和自变量来确定变量之间的因果关系。回归分析与关联规则分析不同，关联规则分析重点在于分析变量之间是否相关，及相关的密切程度；回归分析重点在于分析变量之间是否存在因果性。例如，从关联规则分析得知 A 和 B 变量密切相关，但是这两个变量之间是谁影响谁，影响程度如何，则需要通过回归分析方法来确定。回归分析按照涉及自变量的多少，可分为一元回归分析和多元回归分析；按照自变量和因变量之间的关系类型，可分为线性回归分析和非线性回归分析。下面介绍在大数据中经常使用的两种回归分析算法——局部加权线性回归（Locally Weighted

Linear Regression)和主成分回归分析法(Principal Components Analysis)。

1) 局部加权线性回归

局部加权线性回归算法是由 Cleveland 提出,后来由 Cleveland 和 Develin 推广到多个自变量的情形,该算法主要根据已有的训练样本,对已选择的特征进行多项式加权拟合并计算出权重值,并根据计算出的权重值利用最小二乘法来估算预测结果。该算法每次预测都要根据原有的训练样本去重新学习计算权值,这样使得计算代价比较高。

2) 主成分回归分析法

主成分回归分析法是一种把多指标转化为少数几个综合指标的降维思想的回归分析方法。在实证问题的研究中,为了全面、系统地分析问题,必须考虑众多因素,而这些涉及的因素一般称为指标,每个指标都在不同程度上反映了所研究问题的某些信息,并且指标之间彼此有一定的相关性。这些指标太多会增加计算量和增加分析问题的复杂性,这种情况就可以通过主成分回归分析法来选择合理指标,从而有助于问题的分析。

2.3.2　大数据分析与挖掘方法

随着海量半结构化和非结构化数据的迅速增长,给传统的数据分析与挖掘方法带来了冲击和挑战。因为传统数据分析与挖掘方法大多都是以数据量小为前提的、基于内存基础上所构造的算法,而面对大数据时,就需要保证具有基于外存以及处理大规模数据集合的能力。下面主要针对大数据的常用分析与挖掘方法进行简单举例说明,虽然这些方法并不能完全覆盖大数据分析所要处理的所有问题,但是可以处理大数据分析所面临的一些共性问题。

1. 布隆过滤器

布隆过滤器(Bloom Filter)[54]是 Howard Bloom 在 1970 年提出的一种多哈希函数映射的快速查找算法,具有很好的空间和时间效率,被用来非常快速地判定某个元素是否在一个集合之外。这种检测只会对在集合内的数据错判,而不会对不是集合内的数据进行错判,即每个检测请求返回只有“在集合内(可能错)”和“不在集合内(绝对不在集合内)”两种情况。

布隆过滤器的基本思想是:当一个元素被加入集合时,通过 K 个 Hash 函数将这个元素映射成一个位阵列(Bit Array)中的 K 个点,并把这些点置为 1,检索时,如果发现所有 Hash 函数对应位都是 1,说明被检索元素很可能在集合中;如果发现所有 Hash 函数对应位有任何一个不为 1,则说明被检索元素一定不在集合中。该算法的优点就是插入和查询时间都是常数。另外,查询元素时却不保存元素本身,具有良好的安全性。缺点也是当插入的元素越多,错判“在集合内”的概率就越大了。另外,布隆过滤器也不能删除一个元素,因为多个元素哈希的结果可能在布隆过滤器结构中占用的是同一个位,如果删除了一个比特位,可能会影响多个元素的检测。该算法通过极少的错误换取了存储空间的极大节省。因此,适合于能容忍低错误率的应用场合布隆过滤器,不适合那些“零错误”的应用场合。目前,布隆过滤器被广泛应用在分布式系统中,用来判断某个元素是否存在于

大数据量的集合中，如布隆过滤器应用在 GFS\HDFS\Cassandra\Bigtable 等分布式文件系统中，以减少不存在的行或列在磁盘上的查询，从而大大提高了数据库的查询操作性能。

2. 哈希算法

哈希算法将任意长度的二进制值映射为较短的固定长度的二进制值，用来加快查找的速度，是一种典型的“空间换时间”的快速存取角度设计的算法。这个小的二进制值称为哈希值，映射函数叫做散列函数，存放哈希值的数组叫做散列表。哈希算法本质上就是根据关键码值(Key/Value)直接进行访问的数据结构，把关键码(Key)通过一个固定的算法函数(即哈希函数)转换成一个整型数字，然后将该数字对数组长度进行取余，取余结果作为数组的下标，将值(Value)存储在以该数字为下标的数组空间中。当使用哈希表进行查询操作时，就再次使用哈希函数将关键码(Key)转换为对应的数据下标，并定位到该空间获取值(Value)。哈希算法的这种将任意长度的输入经过映射后得到固定长度的输出，是一种单向不可逆的映射过程。因此，哈希算法一般用于快速查找和加密算法，常见的哈希算法包括 MD2、MD4、MD5 和 SHA-1 等。

当处理海量的数据时，可以采用分布式哈希算法，该算法的核心思想是采用分而治之的方法，将海量数据切分为若干份来进行处理，并且在处理的过程中要兼顾内存的使用情况和处理并发量以及一致性等情况。

3. 字典树

字典树(Trie Tree)是一种树状结构，是一种哈希树的变种，其核心思想是“通过空间换时间”，利用字符串的公共前缀来降低查询时间的开销以达到提高效率的目的。该算法的基本思想与字典的查询过程很相似，当要查一个单词是不是在字典树中时，首先看该单词的第一个字母是否在字典的第一层，如果不在，说明字典树里没有该单词；如果在，就在该字母的孩子节点里查找是否有单词的第二个字母，如果没有，说明没有该单词；有的话，继续用同样的方法查找，直到遍历完整个树为止。字典树不仅可以用来存储字母，也可以用来存储数字等其他数据。该算法的典型应用是用于统计和排序大量的字符串(但不仅限于字符串)，所以经常被搜索引擎系统用于文本词频统计。它的优点是最大限度地减少无谓的字符串比较，查询效率比哈希表高。

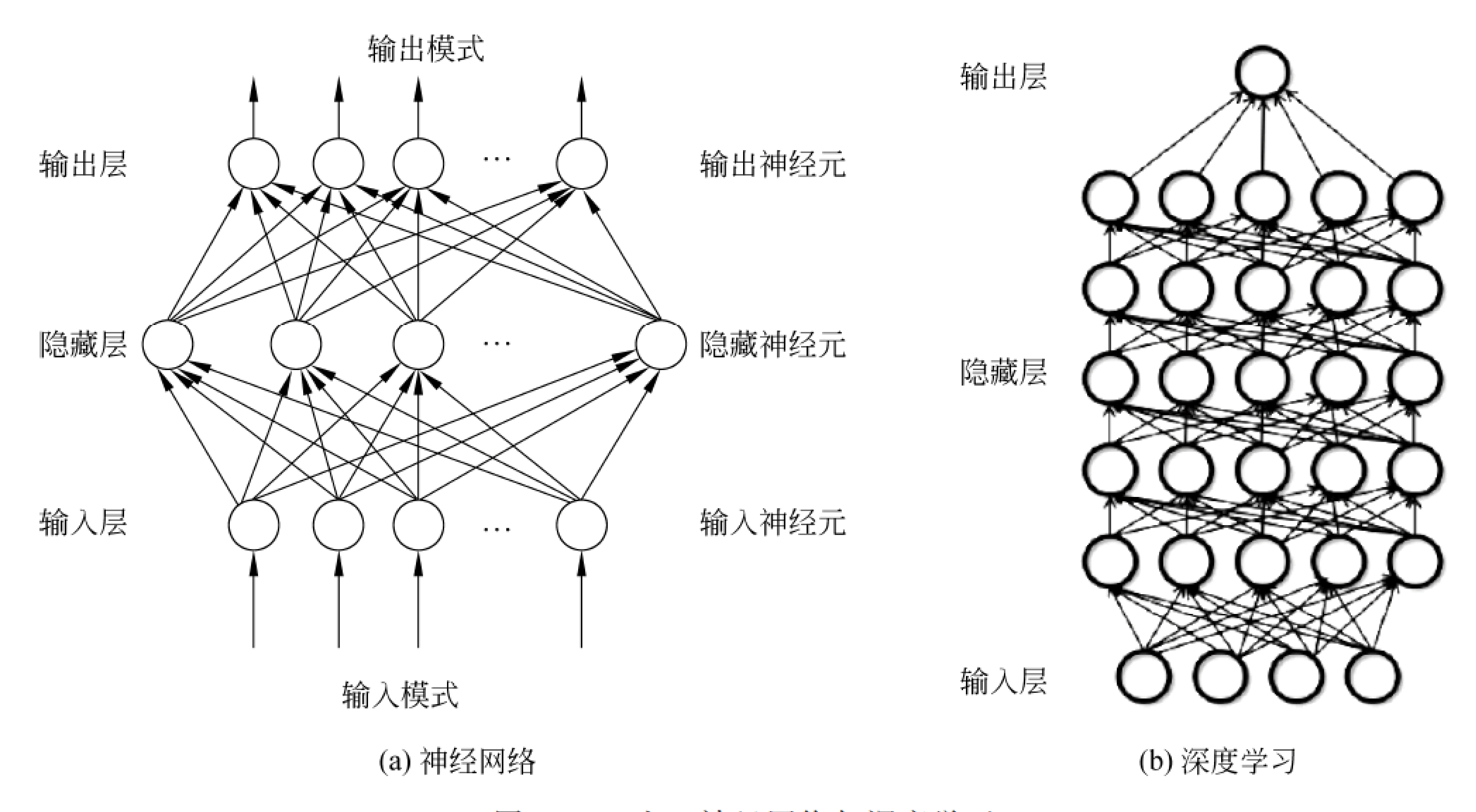
4. 深度学习

大数据分析与挖掘技术主要是对海量的结构化和非结构化数据进行高效的深度分析，挖掘内在隐性知识，如从网页数据中理解和识别语义、情感等，把海量复杂多源的语音、图像、视频数据转化为机器可识别、具有明确语义的信息等。如果采用传统的机器学习方法，首先要通过先验知识人工建立数据模型来分析数据，并使用大量样本数据进行训练，从而获得从数据中提取知识的能力。然而，对于非结构化数据而言，靠人工建立分析数学模型的传统机器学习方式很难挖掘出隐藏在数据中的隐含知识。因此，通过人工智



能和机器学习技术相结合使得机器学习领域取得了突破性的进展，即深度学习[55-57]。深度学习是一种基于无监督特征学习和特征层次结构的学习方法，是机器学习研究中的一个新领域，其动机在于建立、模拟人脑进行分析学习的神经网络，通过模仿人脑的机制来解释数据。

深度学习本质上是一个人工神经网络，两者都包含输入层、输出层及中间的若干隐藏层组成的多层网络，每层都有若干节点及连接这些点的边，同一层以及跨层节点之间相互无连接。每一层可以看作是一个逻辑回归模型，在训练数据集上会学习出边的权值，从而建立模型。两者的区别在于深度学习是含多个隐藏层的多层感知器（人工神经网络虽然也称作多层感知器，但实际是一种只含有一层隐藏层节点的浅层模型），通过组合低层特征形成更加抽象的高层表示属性类别或特征，以发现数据的分布式特征表示，如图 2.11 所示。



(a) 神经网络　　(b) 深度学习

图 2.11　人工神经网络与深度学习

图 2.11 给出了神经网络与深度学习的区别，传统的神经网络采用的是反向传播（BP 机制）的方式进行，即采用迭代的算法来训练整个网络，随机设定初值，计算当前网络的输出，然后根据当前计算的输出值和实际的标记值之间的差去改变前面各层的参数，直到收敛。而深度学习采用逐层训练机制（BP 机制不适合深度学习，因为对于一个 7 层以上的深度网络，残差传播到最前面的层将变得很小，会出现所谓的梯度扩散），强调了模型结构的深度（通常有 5～10 层以上的隐藏节点），明确突出了特征学习的重要性，即通过逐层特征变换，将样本在原空间的特征表示变换到一个新特征空间，从而使分类或预测更加容易。利用大数据来学习特征，更能够刻画数据的丰富内在信息。目前，深度学习技术已经在语音识别和图像识别方面取得了很好的效果，如果要在大数据分析上广泛应用，还有大量理论和工程问题需要解决。

2.3.3 大数据分析与挖掘框架

面对海量的结构化、半结构化和非结构化数据种类和类型的日益增多，传统的结构化数据分析与挖掘框架已无法面对数据量达到 PB、EB 或 ZB 级别的数据，如数据获取、存储、检索、分析和可视化等。因此，在面对“大数据”时，需要新的分析框架才能满足海量数据的获取、存储、分析和可视化，才能具有更强的决策力、洞察发现力等。根据大数据处理多样性和分析需求驱动，产生了适合于大数据批处理的并行计算框架，如 Hadoop MapReduce、UCBerkeley Spark(具备批处理计算能力)等；具备高实时性的流式计算框架，如 Twitter Storm、Apache S4、Apache Spark Steaming、Apache Samza 等；具有快速和灵活的迭代计算框架，如 UCBerkeley Spark、HaLoop Twister 等；具备复杂数据关系图数据的分析框架，如 Google Pregel、Facebook Giraph、Microsoft Trinity、Spark GraphX、PowerGraph 等；具备实时内存计算能力的大数据分析框架 SAP Hana、UCBerkeley Spark 等。这几种大数据分析框架各具特点，都起源于某个特殊的应用领域，表 2.2 给出了各分析框架的特点及具体应用领域。

表 2.2　各分析框架特点及应用领域

分　类	名　称	特　点	应用领域
批处理框架	MapReduce UCBerkeley Spark	高度可扩展、高容错能力、动态灵活的资源分配	数据分析、日志分析、数据挖掘、机器学习等
流式计算框架	Twitter Storm Apache S4 Spark Steaming Apache Samza	保证响应时间的事务功能、消息精确处理、动态流数据处理、记录级容错	在线机器学习、连续计算、数据采集等
迭代计算框架	UCBerkeley Spark HaLoop Twister	循环控制、数据缓存、减少磁盘 I/O	数据挖掘、信息检索、实时视频处理等
内存计算框架	SAP Hana UCBerkeley Spark	基于内存计算的可扩展集群、交互式数据处理、实时返回分析结果	实时分析、数据挖掘、机器学习、可视化模式分析等
图计算框架	Google Pregel Facebook Giraph Microsoft Trinity Spark GraphX PowerGraph	基于 BSP(Bulk Synchronous Parallel)模型的分布式图计算框架、占用较低资源的消息通信机制、同步控制框架	矩阵计算、面向图计算、排序计算、图索引、PageRank 等

从表 2.2 可以看出，批处理框架适用于对反馈时间要求不是那么严苛的离线数据分析，如离线统计分析、机器学习、搜索引擎的反向索引计算、推荐引擎的计算等；流式计算框架主要适用于准实时数据分析，如金融领域的风险管理、商业智能、精准推荐等；迭代计算框架和内存计算框架是一种交互式数据分析框架，主要适用于对反馈时间要求比较严苛的实时交互数据分析，如在线实时服务、实时视频分析、实时日志分析等；图计算框架适用于互联网应用中的 PageRank、排序、社会网络结构分析等。下面对这几种不同类型的大数据分析与挖掘框架所处理的数据形式和特征、各自典型的应用及代表性的框架进行

详细介绍。

1. 批处理

批处理分析框架是一种高性能的批处理分布式计算框架，主要用于处理海量的结构化、半结构化和非结构化数据，该分析框架适用于先存储后计算，对实时性要求不高，对数据的准确性和全面性更为重要的场景。

1）批处理数据的特征

批处理的数据通常具有数据量大、数据精确度高和数据价值密度低的特性。数据量大主要体现在数据在 TB 级别和 PB 级别这个量级上，并且数据长期以静态形式存储在磁盘中，很少进行更新等操作；数据精度高主要体现在批处理的数据往往是从实际的业务应用中沉淀下来、具有很高的商业或应用价值的数据，已成为企业资产的一部分宝贵财富；数据价值密度低主要体现在批处理的数据往往具有高维度、低密度的特性，如以视频批处理数据为例，在连续不间断的监控过程中，可能有用的数据仅有一两秒。

2）典型应用

批处理分析框架可用于分布排序、Web 访问日志分析、反向索引构建、文档聚类、机器学习、基于统计的机器翻译等对实时性要求不高的大规模数据处理工作。在互联网领域中，通过批处理分析框架可进行社交网络的分析，如在 Facebook、微博、微信等以人为核心的社交网络中所产生的大量的文本、图片、音视频等多种类型的海量数据进行批处理分析，从而发现人与人之间隐含的关系等，推荐给朋友或相关主题，提升用户的体验。在电子商务领域中，如淘宝网拥有国内最具商业价值的海量交易数据，其主要活动角色包括卖家、供应商和买家，通过搜索、浏览、收藏、交易、评价等产生上千万的成交、收藏和评价数据，通过批处理分析这些数据，可以精准地选择其热卖商品，从而提升商品销量，还可以通过批处理分析出用户的消费行为，为客户推荐相关商品，从而挖掘出真正的商业价值，帮助淘宝、商家进行企业的数据化运营，帮助消费者进行理性的购物决策。

3）具有代表性的框架

批处理分析框架最具有代表性的就是 MapReduce 编程模型，该框架采用无共享大规模集群系统，具有良好的性价比和可伸缩性，适合处理各种类型的数据，如结构化、半结构化和非结构化数据，处理的数据量都在 TB 和 PB 级别。MapReduce 框架将所处理的任务分为大量的并行 Map 任务和 Reduce 汇总任务两类，具体处理过程是将任务分发给一个主节点管理下的各个分节点共同完成（Map 过程），然后通过整合各个节点的中间结果，得到最终结果（Reduce 过程），如图 2.12 所示。因此，用 MapReduce 来处理的数据集必须具备数据集可以分解成许多小的数据集，而且每个小的数据集都可以完全并行地进行处理的特点。

2. 流式数据分析

对于无须事先存储，可直接进行数据计算，对实时性要求比较高，但对数据的精确度要求稍微宽松的应用场景时，流式数据分析框架具有明显的优势，使用该框架更为合适。流式数据分析框架要求数据延迟较短，实时性较强，但对数据精确度要求往往较低，这与

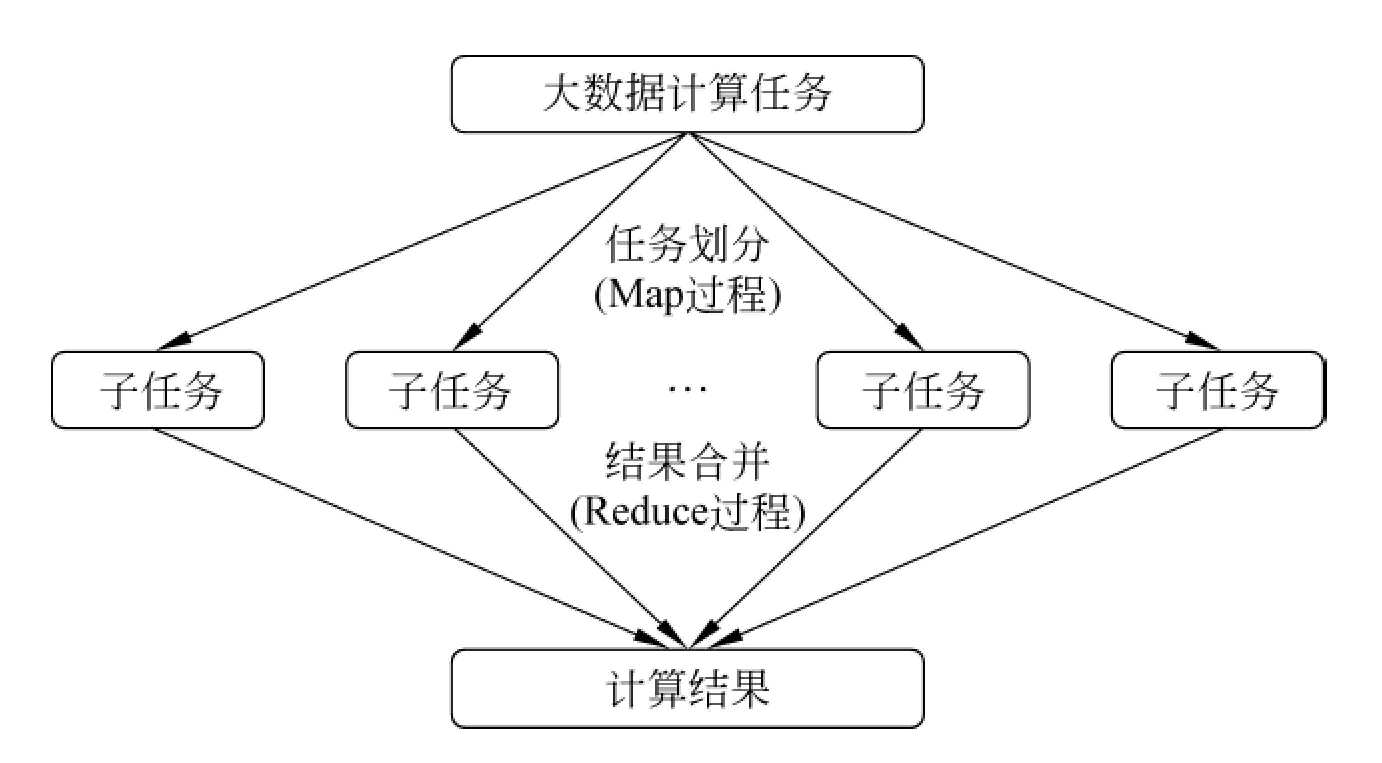


图 2.12 MapReduce 框架的任务划分和并行计算模型

批处理分析框架具有明显的优劣互补特征。在实际应用时,可以将两者结合起来,通过发挥流式数据分析的实时性优势和批处理分析的精确性优势来满足多种应用场景在不同阶段的数据计算要求。

1）流式数据的特征

流式数据分析框架所处理的数据通常具有实时性、易失性、突发性、无序性、无限性等特性。流式数据的实时性主要体现在实时产生、实时计算、数据价值的有效时间往往较短、结果反馈往往也需要保证及时;易失性主要体现在数据流到达后立即被计算并使用,只有极少数的数据才会被持久化地保存下来,其余的数据会被直接丢弃;突发性主要体现在数据的产生完全由数据源确定,不同的数据源在不同时空范围内的状态不统一且极易发生动态变化;无序性主要体现在各数据流的到达顺序是不可预知的,无法保证新数据流与之前数据流中的数据元素顺序是一致的;无限性主要体现在数据是实时产生的、动态增加的,只要数据源处于活动状态,数据就会一直产生和持续增加下去。

2）典型应用

目前,流式数据分析框架主要应用于数据采集、搜索引擎、广告精准推荐、商业智能、金融领域的风险管理、社交网络、智能交通等方面。在数据采集应用方面,通过流式数据分析框架主动获取海量的实时数据,及时地挖掘出有价值的信息,如 Facebook 的 Scribe、LinkedIn 的 Kafka、Cloudera 的 Flume 等。在搜索引擎方面,通过流式分析框架对引擎使用者的查询偏好、浏览历史、地理位置等综合信息进行分析,从而决定在搜索页面中要插入什么广告、在什么位置插入这些广告才能得到最佳效果等。在金融领域方面,通过流式数据分析框架可以对日常运营过程中产生的大量具有时效性的结构化、半结构化和非结构化数据进行流式分析,发现隐藏于其中的内在特征,可以帮助金融银行进行信用卡诈骗检测、保险诈骗检测、交易诈骗检测、个性化推荐服务等。

3）具有代表性的框架

流式数据分析框架已在业界得到广泛应用,具有代表性的有 Twitter 的 Storm、Yahoo 的 S4（Simple Scalable Streaming System）、Facebook 的 Puma、Hadoop 的 HStreaming、Spark Streaming 等。Storm 是一个免费开源、分布式、高容错的流式计算系统,经常用于在线实时分析、在线机器学习、持续计算、分布式远程调用和 ETL 等领域。

Yahoo 的 S4 是一个通用的、分布式、可扩展、分区容错、可插拔的流式系统，Yahoo 开发 S4 主要是为了解决搜索广告的展现及处理用户的点击反馈。Facebook 使用 Puma 和 HBase 相结合来处理实时数据，使批处理计算平台具备一定实时能力。HStreaming 是 Hadoop 的一个实时组件，能让 Hadoop 平台具备实时数据处理能力。Spark Streaming 是建立在 Spark 上的应用框架，利用 Spark 的底层框架作为其执行基础，并构建了 DStream 的行为抽象，用户可以在数据流上进行实时分析操作。

3. 交互式数据分析

对于存储在系统中的数据文件能够被及时处理修改，同时处理结果可以立刻被使用的这种交互式的应用场景时，交互式数据分析框架则具有明显优势。其中，迭代计算框架和内存计算框架都是一种交互式数据分析框架。

1）交互式数据的特征

与非交互式数据相比，交互式数据处理更为灵活、直观、便于控制，数据以对话的方式输入，系统便提供相应的数据或者提示信息，引导其逐步完成所需的操作，直至获得最后处理结果。

2）典型应用

目前，交互式数据分析框架主要应用于需要人机交互并实时反馈结果的应用场景，如搜索引擎、电子邮件、即时通信、社交网络、微博、博客等。用户可以在这些平台上获取或分享各种信息，并与平台之间进行相应的数据交互。

3）具有代表性的框架

交互式数据分析框架具有代表性的有 Berkeley 的 Spark。Spark 是一个基于内存计算的、可扩展的开源集群计算系统，是基于 MapReduce 算法实现的分布式计算，拥有 MapReduce 所具有的优点，但不同于 MapReduce 的是 Job 中间输出和结果可以保存在内存中，从而不再需要读写 HDFS，适用于需要多次操作特定数据集的快速处理、实时返回分析结果的应用场合。

4. 图数据分析

针对图自身的结构特征，通过分析图中点和边的强关联性，可以很好地得到事物之间的关系。但是，随着图中节点和边数的增多，图数据处理的复杂性越来越大，而且难以将图分割成若干完全独立的、可并行处理的问题时，图数据分析框架具有明显的优势。

1）图数据的特征

图数据的种类繁多，如生物、化学、计算机视觉、模式识别、社会网络、知识发现、情报分析等领域的数据都可使用图数据来表示。图数据中包括节点及连接节点的边，并且数据之间是相互关联的。

2）典型应用

图能够很好地表示各实体之间的关系。因此，在各个领域得到了广泛的应用，如自然科学领域、交通领域、互联网领域的应用等。在互联网领域中，Facebook、Twitter 等新兴服务利用图数据分析框架建立了大量的在线社会网络关系，用图来表示人与人之间的关

系；在交通领域中，通过图数据分析框架可在动态网络交通中查找最短路径等。

3）具有代表性的框架

由于不同领域的图数据的处理需求不同，因此，没有一个通用的图数据分析框架可以满足所有领域的需求。当前主要的图数据分析框架有 Google Pregel、Facebook Giraph、Microsoft Trinity 等。Pregel 是 Google 提出的基于 BSP（Bulk Synchronous Parallel）模型的分布式图计算框架，主要用于图遍历（BFS）、最短路径（SSSP）、PageRank 计算等。Giraph 是基于批量同步并行图计算模型，可以实现对不同实体间的万亿个连接进行分析。Trinity 是 Microsoft 推出的一款建立在分布式云存储上的图计算模型，可以并发执行 PageRank、最短路径查询、频繁子图挖掘以及随机游走等操作。

2.4 大数据应用与展现技术

大数据应用与展现技术是利用大数据分析与挖掘的结果，为用户提供辅助决策，发掘潜在价值的过程。大数据应用与展现技术一定要与领域知识相结合，在不同的领域、不同的应用需求下，大数据的获取、分析和展现方式都不同。因此，大数据应用与展现技术的研究需要开展数据特征和业务特征的研究，需要开展大数据的应用分类和技术需求分析。

2.4.1 大数据应用

目前，大数据应用目前朝着两个方向发展，一种是以盈利为目标的商业大数据应用，另一种是不以盈利为目的，侧重于为社会公众提供服务的大数据应用。商业大数据应用主要是以 Facebook、Google、淘宝、百度等公司为代表，以自身拥有的海量用户信息、行为、位置等数据为基础，提供个性化广告推荐、精准化营销、经营分析报告等；公共服务的大数据应用如搜索引擎公司提供的诸如流感趋势预测、春运客流分析、紧急情况响应、城市规划、路政建设、运营模式等方面得到广泛应用。大数据应用基本呈现互联网领先并引领大数据应用，其他行业的大数据应用仍在探索之中。

下面通过一个具体的大数据应用案例来说明大数据在不同的领域、不同的应用需求下，如何开展大数据应用。阿里巴巴金融是互联网金融领域的一个典型大数据应用案例，通过掌握的企业交易数据，借助大数据技术自动分析判定是否给予企业贷款，全程不会出现人工干预。阿里巴巴金融主要有两种模式，“阿里小贷”（为阿里巴巴上的企业客户提供信用贷款）和“淘宝小贷”（为淘宝和天猫客户提供的订单贷款和信用贷款）分别针对不同的客户类型，采取不同的贷款方式。这两种贷款方式，都不需要借款人提供抵押品或第三方担保，仅凭自己的信誉就能取得贷款，这两种贷款方式更注重的是数据而不是担保和抵押。截至目前，阿里巴巴已经放贷三百多亿元，坏账率约 0.3%左右，大幅度低于商业银行。

从图 2.13 可以看出，阿里巴巴小贷建立了多层次微贷风险预警和管理体系，通过大数据技术实现了贷前、贷中和贷后的一体化数据采集和分析，有效规避和防范贷款风险。如在贷前根据企业（个人）电子商务经营数据和第三方认证数据，辨析企业经营状况，反映企业（个人）偿债能力；贷中通过支付宝及阿里云平台实时监控商户（个人）的交易状况和

现金流，为风险预警提供信息输入；贷后通过大数据技术监控企业（个人）经营动态和行为，可能影响正常履约的行为将被预警，从而提高客户违约成本，有效控制贷款风险。阿里巴巴之所以能成功地介入到金融服务领域，核心优势就是拥有庞大的客户资源和数据，并能基于云计算平台通过大数据技术对客户信息的充分分析、挖掘，实现对客户信用水平和还款能力的准确、实时的把控。

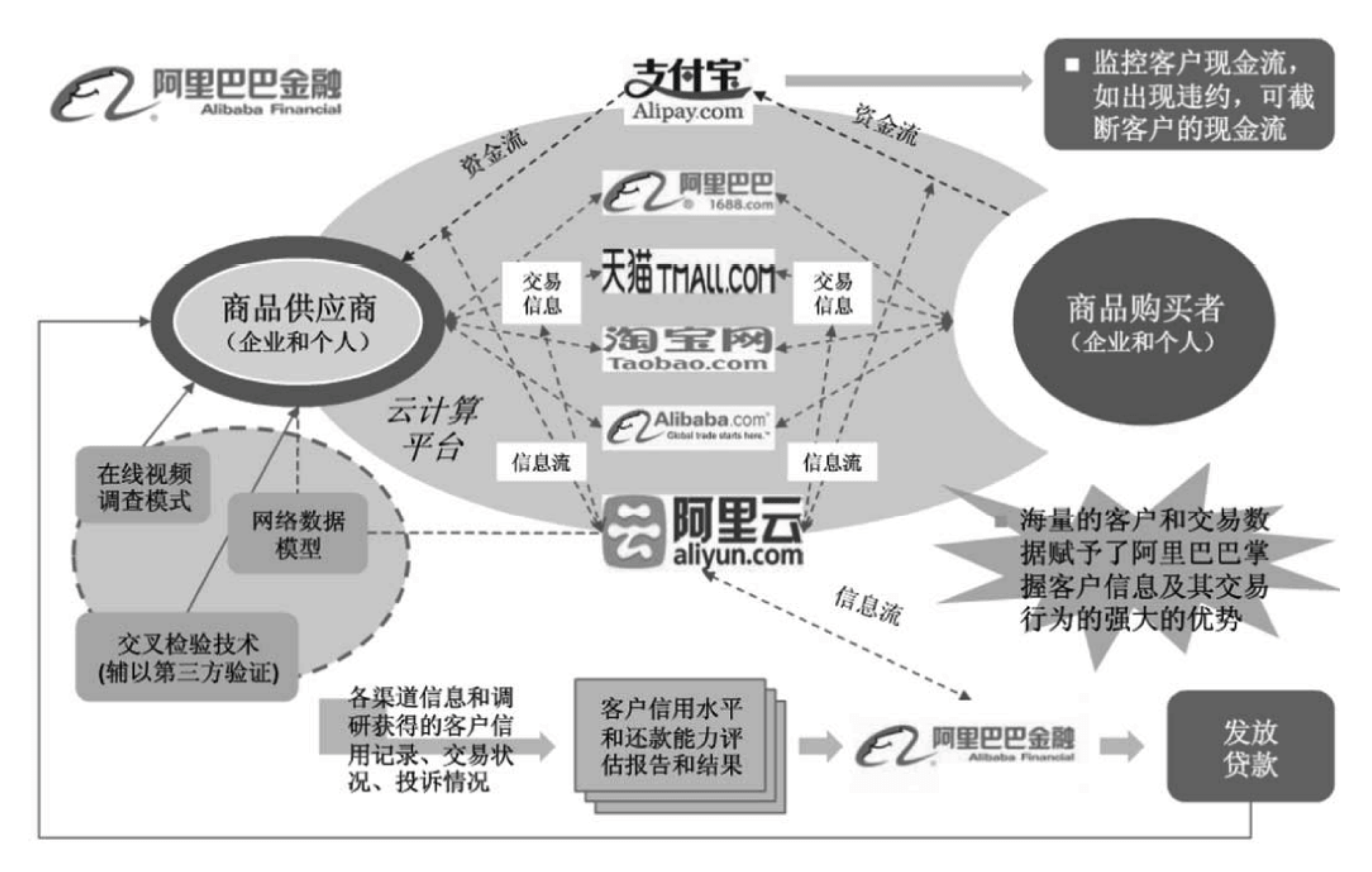


图 2.13　阿里巴巴金融创新大数据应用模式

通过深入分析互联网企业大数据应用的成功案例，可以总结出大数据应用所必备的以下几个条件。

(1) 丰富的数据和强大的平台是基础条件。如 Google 的数据资源极为丰富，拥有全球网页搜索索引库，掌握几十亿用户的搜索行为数据，其中包括用户行为数据、网页数据、系统日志数据、用户交易数据等，映射出各种大数据的创新应用，如定向广告、个性推荐、机器翻译等较成熟的大数据应用。

(2) 应用总不是飞跃性的，要靠获取长期的效益积累。如搜索引擎、广告、推荐等成熟应用都是在日积月累的微小进步中逐渐形成的，其中搜索引擎是最早的互联网大数据应用，如 Google、百度等提供搜索引擎。

(3) 积累效益的获取，要靠不断的技术迭代。互联网企业一直奉行敏捷开发、快速迭代的软件开发理念，可以在很短的周期内完成一个"规划、开发、测试、发布"的迭代周期。这种长期持续"小步快跑"的研发方式，支撑大数据应用效果的持续提升，并建立了技术的领先优势。

(4) 快速迭代的保障要通过技术和应用一体化组织。互联网企业之所以能够保持高效率的持续技术演进，其研发和应用一体化的组织方式是很重要的一个因素，从而形成一

套高效运转的研发产业化体系,并能够启发新的创意,加快再创新步伐。

2.4.2 大数据可视化

传统的数据可视化只是将数据加以组合,通过不同的展现方式提供给用户,用于发现数据之间的关联信息。在大数据背景下,传统的数据可视化分析模型、理论已无法满足需求,必须针对大数据的海量性、实时性、价值性等特点重新构建一套有效的可视化分析理论及分析模型。因此,大数据可视化所面临最大的挑战就是如何提出新的可视化方法能够帮助人们分析大规模、高维度、多来源、动态演化的数据,并辅助做出实时的决策。目前,大数据可视化应用也更加广泛并衍生了许多新的研究方向,主要研究方向有数据可视化、科学计算可视化、信息可视化、知识可视化等。

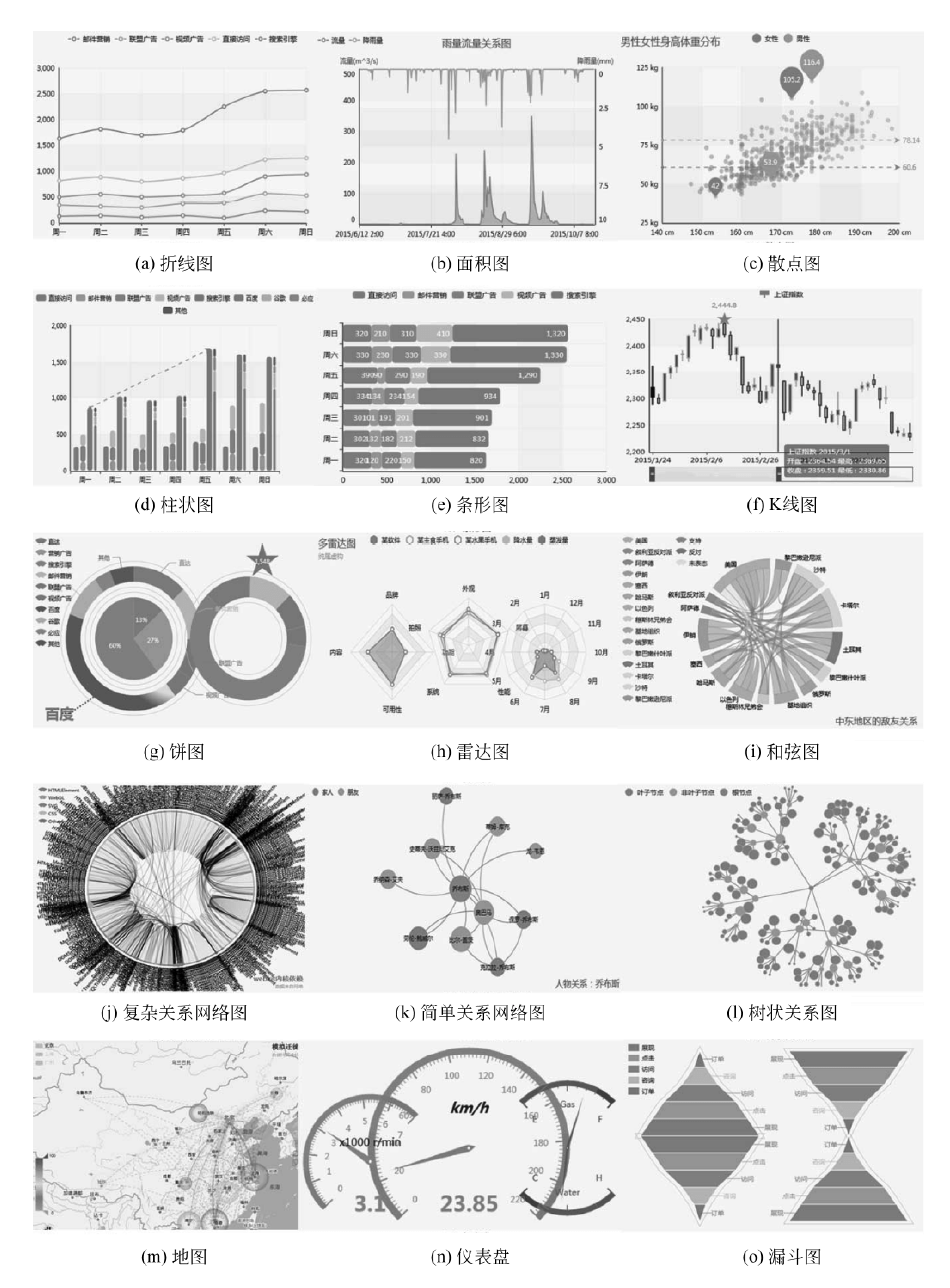
1. 数据可视化

数据可视化主要面向的是大型数据库中的数据,借助于图形化手段,如折线图、柱状图、散点图、饼状图、地图、网络图、雷达图、矩阵图等直观地表达数据与数据之间的关系,获得数据内在的信息,从而清晰有效地传达与沟通信息。面对大数据的海量、异构、多样性等特征的数据集,如商业分析、人口状况分布、用户行为数据等,数据可视化可能要经历包括数据采集、数据分析、数据治理、数据管理、数据挖掘在内的一系列复杂数据处理,然后根据业务需求的场景来确定所采用的图形化方式,例如采用立体的还是二维的、静态的还是动态的、实时的还是交互式的图形化表现形式等。

数据可视化也是根据需求以及数据维度或属性进行筛选,根据目的和用户群选用不同的表现方式。即使是相同的数据,也可以可视化成多种看起来截然不同的形式,比如有的可视化目标是为了观测、跟踪数据,有的是为了分析数据,有的是为了发现数据之间的潜在关联,有的是为了帮助用户快速理解数据含义或变化等。因此,如何对海量复杂的数据集进行直观、生动、可交互的解释,其本身就是一门艺术。图 2.14 列举了一些常用的数据可视化的图形表现形式,如折线图、柱状图、散点图、K 线图、饼图、雷达图、和弦图、力导向布局图、地图、仪表盘、漏斗图等,读者可根据自己的实际需求进行参考。

2. 信息可视化

信息可视化主要面向的是大规模非数值型信息资源,即本身没有几何属性和明显空间特征的抽象的、非结构化的数据集合,如文本信息、语音信息、视频信息等,利用图形图像方面的技术与方法,将抽象数据用可视的形式表示出来,帮助人们理解和分析数据,从而发现数据中隐藏的特征、关系和模式等。信息可视化的关键就是如何将数据用有意义的图形表示出来,其主要过程包括对数据进行描述,再利用可视化方法对数据进行表示,从而挖掘数据信息内在的有用信息,然后利用特征提取、特征优化、模式识别、数据挖掘等手段对信息进行处理,最终增强人们对抽象信息的认识或辅助人们得出某种结论性观点。图 2.15 列举了几种信息可视化的图形表现形式,如图 2.15(a)是对美国人口分布情况的



(a) 折线图　(b) 面积图　(c) 散点图

(d) 柱状图　(e) 条形图　(f) K线图

(g) 饼图　(h) 雷达图　(i) 和弦图

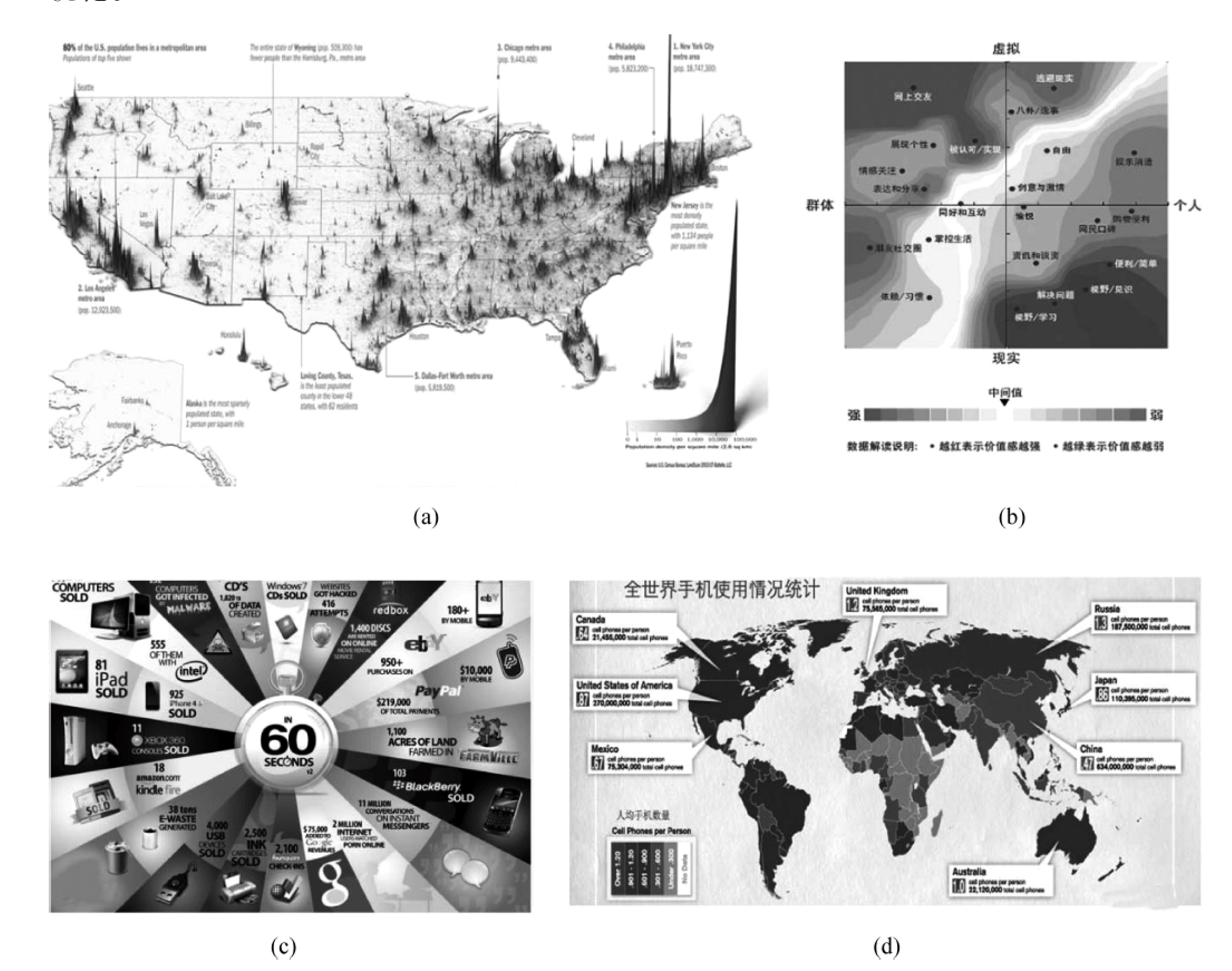
(j) 复杂关系网络图　(k) 简单关系网络图　(l) 树状关系图

(m) 地图　(n) 仪表盘　(o) 漏斗图

图 2.14　常用的数据可视化的图形表现形式

(注：图表中的数据均为模拟数据)

信息可视化；图 2.15(b)是对一至四线城市网民互联网价值的信息可视化；图 2.15(c)是对互联网 60s 会发生什么的信息可视化；图 2.15(d)是对全世界手机使用情况的信息可视化。



(a) (b)

(c) (d)

图 2.15 几种信息可视化的图形表现形式

3. 科学计算可视化

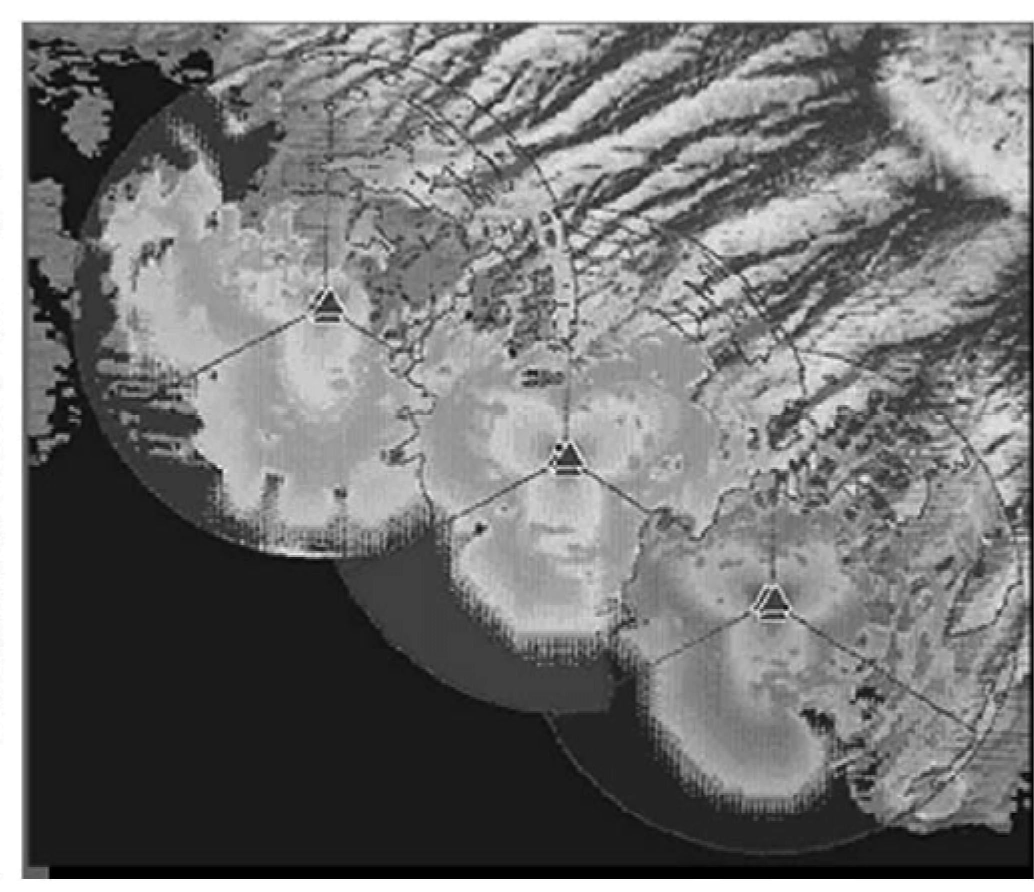
科学计算可视化主要是面向科学及工程测量的、具有几何性质或结构特征的数据，利用计算机图形学、图像处理等技术，将科学数据中所蕴含的现象、规律通过三维、动态模拟等方式表现出来，从而促进人们对数据的洞察和理解。目前，科学计算可视化的主要应用领域有分子建模、计算流体力学、空间探索、医学图像、地理信息、气象、石油、生物信息、有限元分析等，通过对科学数据进行解释和处理来使科学工作者寻找其中的模式、特点、关系等。科学计算可视化的研究重点在于，如何设计和选择合理的显示方式，使用户可以了解海量的多维数据及数据之间的相互关系等，其主要过程包括数据变换、映射、绘制/显示等步骤。其中，变换是对数据进行预处理，如对庞大的数据量只提取与可视目标相关的信息以减少数据量、通过几何变换对点的坐标进行缩放、通过拓扑变换调整网格点的连接关系等；映射是整个科学计算可视化的核心，即设计合理的可视化方案和算法，如二维标量



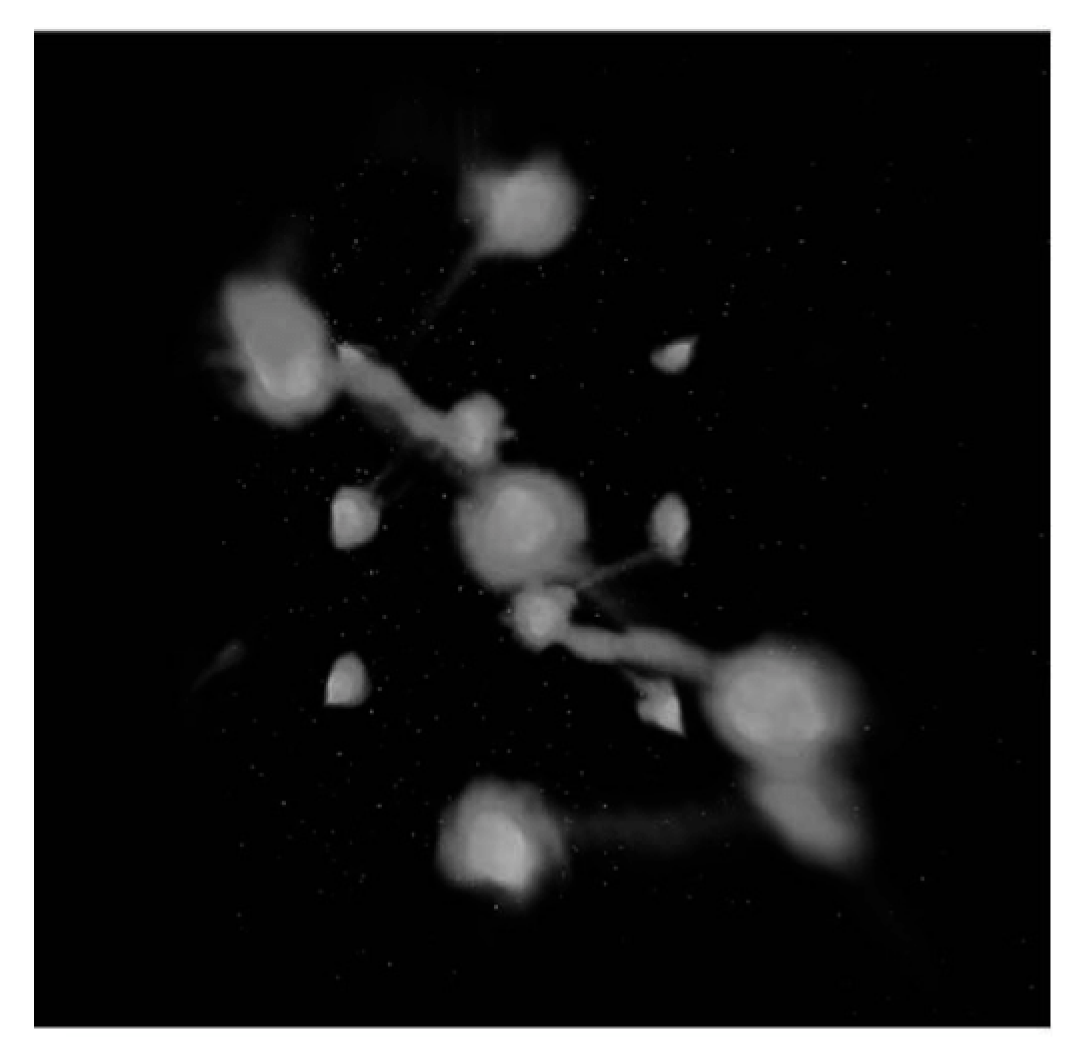
场等值线抽取算法、断层间表面重构算法、等值面生成和绘制算法、体绘制算法等;绘制/显示是科学技术可视化的最后一个步骤,主要是将上述可供绘制的元素转换成图像,绘制在屏幕或其他介质上。图 2.16 列举了几种科学计算可视化的图形表现形式,如图 2.16(a)是对气象预报实时数据的科学计算可视化;图 2.16(b)是对电信覆盖区域数据的科学计算可视化;图 2.16(c)是对天体物理的数据进行科学计算可视化;图 2.16(d)是对化学分子数据的科学计算可视化。



(a) 气象预报实时图

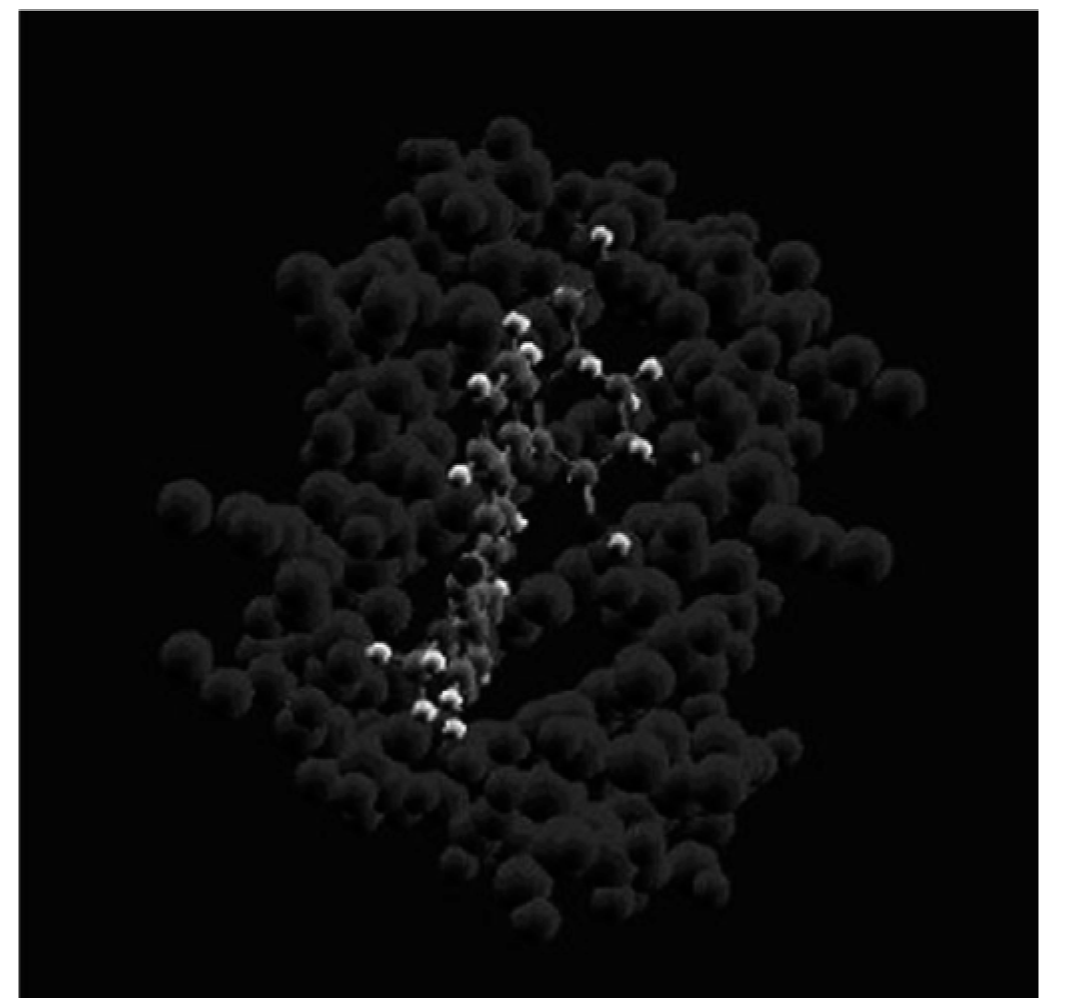


(b) 电信覆盖区域图





(c) 天体物理图



(d) 化学分子图

图 2.16　几种科学计算可视化的图形表现形式

数据可视化与信息可视化的区别在于数据是否为非数值型。目前,对低维、小规模的数值型数据的可视化方法与技术比较成熟,而面对大规模、海量的高维数据时,以前的数据可视化方法就无法满足要求,其处理方法越来越接近信息可视化。因此,对于大数据而言,数据可视化和信息可视化并没有明确的界限。科学计算可视化与数据可视化的主要

区别在于被可视化的对象是物理空间数据还是非物理空间数据。实际上，可以认为科学计算可视化的数据只是数据可视化的一部分而已。

总之，大数据可视化是一门以信息科学、计算机科学、地图学、认知科学、信息传播学、人工智能等学科为基础，并通过计算机技术、数字技术、多媒体技术动态、直观、形象地表现、解释、传递信息并提示其规律的一门综合性的学科。目前，对大数据可视化仍然还处在初始阶段，特别是对于动态多维度大数据流的可视化技术还非常匮乏，需要扩展现有的可视化方法或研究新可视化方法来应对复杂的信息流数据，同时也需要设计创新的交互方式来对大数据进行可视化交互和辅助决策。



基于 Hadoop 的大数据生态系统

狭义的大数据生态系统主要涵盖数据存储与管理、数据安全、数据分析、数据呈现、数据应用等环节;而广义的大数据生态系统则包括数据的整个生命周期,即从数据产生、采集、存储、管理、分析,直至最终的数据展现与应用的过程。Hadoop 作为新一代的架构和技术,具有可扩展、经济、可靠、高效等特性,而且有利于并行分布式处理"大数据",现已成为大数据时代数据处理的首选。目前,基于 Hadoop 的大数据应用已经在互联网领域取得了非常突出的成绩,如 IBM、HP、Intel、EMC、Oracle 均基于 Hadoop 推出大数据商业解决方案,并且 Hadoop 有向电信、电子商务、银行等领域拓展的趋势。因此,Hadoop 已成为企业大数据应用的事实标准。本章内容将以 Hadoop 为基础,介绍基于 Hadoop 的大数据生态系统的发展、架构以及构建过程等内容。

注:很多读者一提到大数据就会想到 Hadoop,认为大数据就是 Hadoop。其实,Hadoop 只是一种分布式系统基础架构,提供了分布式存储系统 HDFS、分布式计算 MapReduce 等技术,从而满足了大数据应用的某方面技术需求。因此,Hadoop 只是大数据的一部分,为大数据研究打开了思路,但绝不代表大数据的全部。

3.1 Hadoop 概述

Hadoop 是以分布式文件系统(Hadoop Distributed File System,HDFS)和 MapReduce(Hadoop 2.0 还包括 YARN)为核心,为用户提供了一个能够对大量数据进行数据挖掘、数据分析、数据存储、数据管理、维护的可靠、高效、可伸缩的分布式基础架构。

3.1.1 Hadoop 发展历程

Hadoop 起源于 Apache 的 Lucene 项目(Lucene 是一个用 Java 编写的文件索引引擎的 API,网址 http://lucene.apache.org)和 Nutch 项目(Nutch 是由 Doug Cutting 通过 Luncene 引擎所开发的一个开源 Web 搜索引擎,网址 http://nutch.apache.org)。当时 Nutch 项目在构建大规模搜索引擎的时候遇到了性能瓶颈,即无法解决数十亿网页的存储和索引问题。于是 Doug Cutting 借鉴了 Google 在 2003 年发表的 GFS 论文 *The*

Google File System[31]和 2004 年发表的 MapReduce 论文 *MapReduce:Simplified Data Processing on Large Cluster*[60]的思想，实现了 Nutch 版的 NDFS 和 MapReduce，从而解决了 Nutch 项目的性能瓶颈。其中，第一篇论文解决了 Nutch 项目遇到的网页抓取和索引过程中产生的超大文件存储问题；第二篇论文解决了处理海量网页数据的索引问题。但 Nutch 项目侧重搜索，而 NDFS 和 MapReduce 则更像是分布式基础架构，故从该项目 Nutch 0.8.0 版本之后，开发人员就把 Nutch 项目中的分布式文件系统 NDFS 以及实现 MapReduce 算法的代码独立出来，形成了一个新的开源项目，并命名为 Hadoop。图 3.1 列出了 Hadoop 发展过程中的一些重要事件。

从图 3.1 可以看出，经过业界和学术界的不断完善发展，Hadoop 已在实际的大数据处理和分析任务中担当着重要角色，并且从 2011 年 Hadoop 1.0.0 发布后，衍生出了多种类型的发行版，如基于社区的 Apache Hadoop 版本、基于开源的 Apache Hadoop 进行改造的商业解决方案（其中包括一系列定制的管理工具和软件）、基于 API 级别和社区 Hadoop 发行版保持兼容的闭源软件等。下面重点介绍基于社区的 Apache Hadoop 版本的发展和基于开源的 Apache Hadoop 进行改造的商业解决方案。

1. 基于社区的 Hadoop 版本演化

基于社区的 Apache Hadoop 版本已由最初的 Hadoop 0.14.1 版本（2007 年 9 月 4 日发行版）进化到如今的 Hadoop 2.6.0 版本（2014 年 11 月 8 日发行版）。图 3.2 给出了 Hadoop 从最初的 0.14 发行版的演化过程。

从图 3.2 中可以看出，从 Hadoop 0.20 版本发布之后，主要功能一直在该分支上进行开发，主干分支并没有合并这个分支。因此，Hadoop 0.20 分支成为主流。Hadoop 0.20.2 版本发布后，有几个重要特性继续在 0.20.2 版本上研发，并衍生出两个主要特性：Append（支持文件追加功能，让用户使用 HBase 的时候避免数据丢失）和 Security（Hadoop 安全机制），而后续的 0.20.205 版本综合了这两个特性，并重命名为 Hadoop 1.0.0 版本。Hadoop 0.21 版本将整个 Hadoop 项目分割成三个独立的模块，分别是 Common、HDFS 和 MapReduce，并包含 Append、Raid、Symlink 和 Namenode HA 特性，但不包括 Security 特性，并在该版本的基础上修复了一些 Bug 并进行了部分优化，衍生出了 Hadoop 0.22 版本。Hadoop 0.20.1 主要应用于商业领域，该版本有一些独特的特性，将在后面的商业版本演化中介绍。表 3.1 给出了 Hadoop 各个版本的特性及稳定性。

从 Hadoop 2.0 版本以后，除以上特性外，还增加了 HDFS Federation 和 YARN 特性。其中，HDFS Federation 支持多个 Namenode 分管没目录，实现访问隔离和横向扩展；YARN 是全新的资源管理框架，将 JobTracker 资源管理（由 ResourceManager 负责）和作业控制（由 ApplicationMaster 负责）功能分开。如果读者想了解 Hadoop 各版本的具体说明，请查看 Apache Hadoop 官方网站（网址 http://hadoop.apache.org/releases.html）。

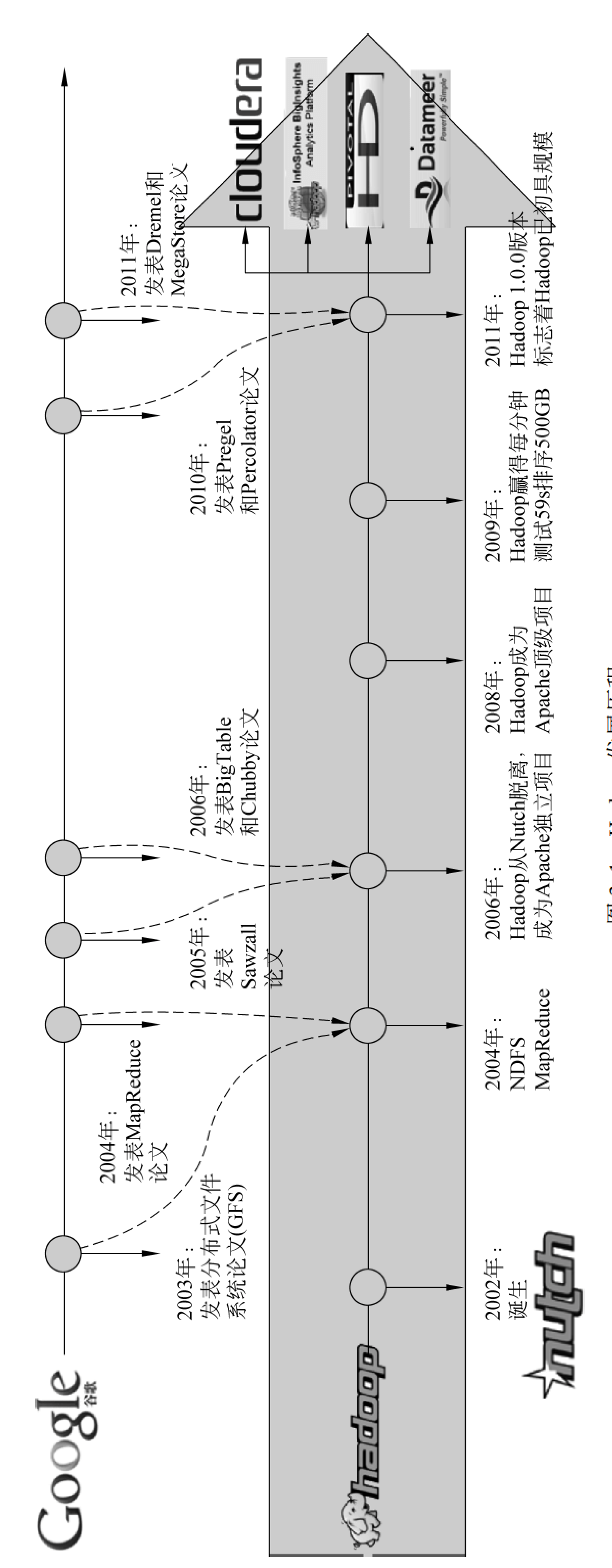


图 3.1 Hadoop 发展历程

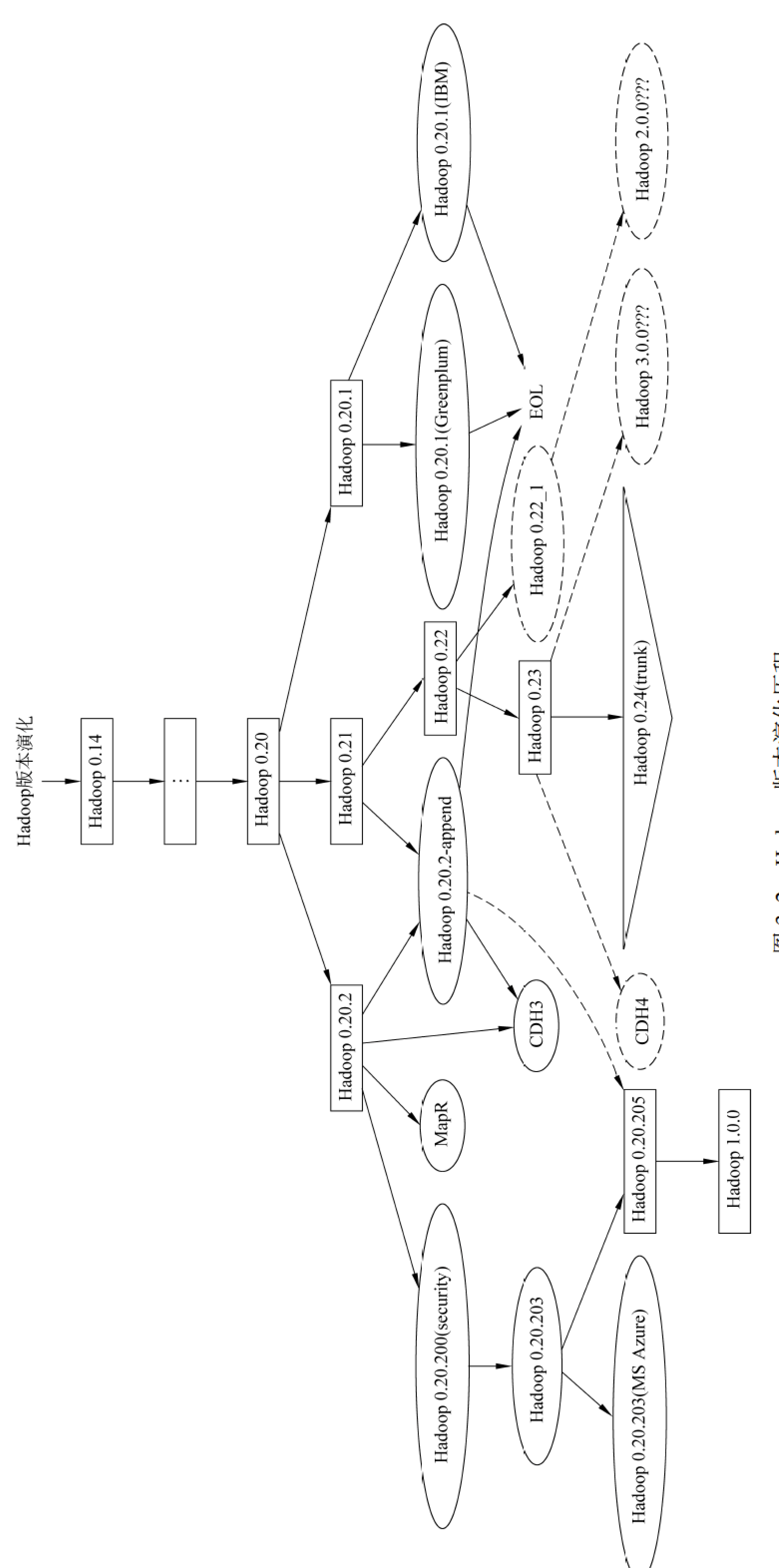


图 3.2　Hadoop 版本演化历程

表 3.1 Hadoop 各版本的特性及稳定性

版本号	特性					是否稳定
	Append	Raid	Symlink	Security	Namenode HA	
0.20.2	×	×	×	×	×	是
0.20.203	×	×	×	√	×	是
0.20.205/1.0	√	×	×	√	×	是
0.21	√	√	√	×	√	否
0.22	√	√	√	√	√	否
0.23.0	√	√	√	√	×	否
1.×	√	×	×	√	×	是
2.×	√	√	√	√	√	否

注：Append：支持文件追加功能，让用户使用 HBase 的时候避免数据丢失。Raid：保证数据可靠，引入检验码校验数据块数目。Symlink：支持 HDFS 文件链接。Security：Hadoop 安全机制。Namenode HA：为了避免 NameNode 单点故障情况，HA 集群有两台 NameNode。

2. 基于开源的 Hadoop 商业版本演化

基于社区的 Hadoop 发行版缺少来自商业供应商的技术支持以及针对自身业务需求完整的解决方案。因此，除了社区的 Apache Hadoop 之外，互联网企业如 Cloudera、Hortonworks、EMC、IBM、Intel、Amazon、微软公司等也分别发布了自己的发行版本。表 3.2 列举了一些 Hadoop 的商业发行版本及不同版本的特点。

表 3.2 Hadoop 商业版本的特性

发行版本	发行方	特点
AWS	Amazon	为大数据应用提供了存储、计算、分析和共享等多方面的支持
CDH	Cloudera	完全开源，比 Apache Hadoop 在兼容性、安全性、稳定性上有增强，并具有管理监控平台
Hortonworks Data Platform (HDP)	Hortonworks	提供数据集成服务、元数据服务、管理和监控服务等
MapR	MapR Technologies	增加了高可用、快照、镜像、通过 NFS 访问数据、控制系统等
Pivotal HD	EMC	与 Greenplum 的兼容性好，适合使用 Greenplum 数据库的用户
InfoSphere BigInsights	IBM	提供了发现数据并加以解释的能力
Apache Hadoop Intel 分发版	Intel	提供全面的软硬件解决方案设计，针对硬件具有更好的性能优化等
Windows Azure HDInsight	Microsoft	主要提供覆盖整个产业链的完整解决方案

3.1.2 Hadoop 特点

Hadoop 是 Apache 基金会下的一个能够对大量数据进行分布式处理的软件框架，该框架最核心的设计是分布式文件系统（Hadoop Distributed File System，HDFS）和 MapReduce。HDFS 有着高容错性的特点，并且设计用来部署在低廉的硬件上，另外 HDFS 能提供高吞吐量的数据访问，适用于大规模数据集的应用程序。MapReduce 是一个编程模型，通过 Map 可将应用程序的工作分解成很多小的工作小块，再通过 Reduce 将所有这些中间的结果合并起来，主要用于处理和生成大规模数据集的相关实现，在处理 TB 级别以上巨量的数据业务上有着明显优势。因此，Hadoop 是以一种可靠、高效、可伸缩的方式进行大规模数据处理的，具体体现在以下几个方面。

1. 高可靠性

高可靠性体现在 Hadoop 能自动地维护多个工作数据副本，并且在任务失败后能自动地重新部署计算任务。

2. 高效性

高效性体现在 Hadoop 以并行的方式处理大规模数据，而且能够在节点之间动态地移动数据，并保证各个节点的动态平衡，从而加快了处理速度。

3. 高可扩展性

高可扩展性体现在 Hadoop 能够通过添加节点获得集群线性性能和容量的提升，可以方便地扩展到数以千计的节点。

4. 低成本

低成本体现在 Hadoop 集群可以由廉价的服务器组成，因此它的成本比较低，而且这些集簇可以方便地扩展到数以千计的节点。

5. 支持多种编程语言

Hadoop 采用 Java 语言编写，因而 Hadoop 支持对 Java 语言编写作业，同时 Hadoop 也支持如 C/C++ 等其他语言编写 MapReduce 作业和非 Java 的第三方库的使用。

3.1.3 Hadoop 核心思想

虽然 Hadoop 仍在不断完善，但其本质上还是基于 Google 的集群系统的开源实现，基本上还是遵循着 Google 发表的一系列论文来完成实现，如 2003 发表的论文 *The Google File System*[31]，该论文描述了如何构造一个分布式的文件系统，能够存储海量数据，并且数据容量能够达到整个互联网所有数据的容量；2004 年发表的论文 *MapReduce:Simplified Data Processing on Large Cluster*[60]，该论文描述了如何在一

个分布式环境下进行海量网页的索引问题，通过分布式编程实现大规模的数据处理，同时不陷入对于系统编程的细节中；2005 年发表的 *Interpreting the data：Parallel analysis with Sawzall*，该论文引入了一个全新的语言 Sawzall，并且提供了一组强力的接口，这些接口属于常用的数据处理和数据合并聚合器；2006 年发表的论文 *A Distributed Storage System for Structured Data*[36] 和 *The Chubby Lock Service for Loosely-Coupled Distributed Systems*[61]，前一篇论文描述了如何在分布式文件系统的基础之上建立用以存储结构化数据的分布式数据库系统，后一篇论文提供了基于 Paxos 实现的一个分布式的锁服务，以文件系统的形式提供编程接口。表 3.3 总结了 Google 发表的几篇论文的对应关系。

表 3.3　Google 与 Hadoop 的对应关系

Google	Hadoop	功　能	论　文
GFS	HDFS	分布式文件系统	*The Google File System*
MapReduce	MapReduce	分布式处理模型	*MapReduce：Simplified Data Processing on Large Cluster*
BigTable	HBase	分布式数据库	*A Distributed Storage System for Structured Data*
Sawzal	Pig，Hive	高级数据流语言	*Interpreting the data：Parallel analysis with Sawzall*
Chubby	ZooKeeper	解决一致性	*The Chubby Lock Service for Loosely-Coupled Distributed Systems*

注：如果读者想深入了解 Hadoop 的核心思想，请认真阅读表 3.3 所给出的 GFS、MapReduce、BigTable、Sawzal 和 Chubby 这几篇论文，相信会有很大帮助。

3.2　Hadoop 家族成员

Hadoop 本身包括 Hadoop Common、HDFS 和 MapReduce（Hadoop 2.0 还包括 Hadoop YARN）。随着 Hadoop 自身不断地完善发展，产生了与 Hadoop 密切相关的数据服务类子项目（如 HBase、Hive、Pig、HCatalog、Sqoop、Flume、Chukwa 等）、运行维护类子项目（如 Ambari、Oozie、ZooKeeper 等）和其他相关类子项目（如 Avro、Mahout 等）。

（1）Hadoop Common（Hadoop 公共服务模块）是 Hadoop 体系最底层的一个模块，为 Hadoop 各子项目提供了开发所需的 API。在 Hadoop 0.20 及以前的版本中，Hadoop Common 包含 HDFS、MapReduce 和其他项目公共内容，从 Hadoop 0.21 开始 HDFS 和 MapReduce 被分离为独立的子项目，其余内容为 Hadoop Common，如系统配置工具 Configuration、远程过程调用 RPC、序列化机制等。

（2）HDFS（Hadoop Distributed File System，分布式文件系统）是一个类似于 Google GFS 的开源的分布式文件系统，是 Hadoop 体系中数据存储管理的基础。它提供了一个可扩展、高可靠、高可用的大规模数据分布式存储管理系统，基于物理上分布在各个数据存储节点的本地 Linux 系统的文件系统，为上层应用程序提供了一个逻辑上成为整体的大规模数据存储文件系统。

(3) MapReduce(并行计算框架)是一种计算模型,用以进行大数据量的计算。其中Map对数据集上的独立元素进行指定的操作,生成键-值对形式的中间结果;Reduce则对中间结果中相同“键”的所有“值”进行规约,以得到最终结果。MapReduce这样的功能划分,非常适合在大量计算机组成的分布式并行环境里进行数据处理。

(4) YARN(Yet Another Resource Negotiator,资源管理框架)是新一代Hadoop资源管理器,用户可以运行和管理同一个物理集群机上的多种作业,例如MapReduce批处理和图形处理作业。它可以对集群中的各类资源进行抽象,并按照一定的策略将资源分配给应用程序或服务。

(5) HBase(分布式列存储数据库)是一个针对结构化数据的可伸缩、高可靠、高性能、分布式和面向列的动态模式数据库。HBase主要用于对大规模数据的随机、实时读写访问,并且HBase中保存的数据可以使用MapReduce来处理,它将数据存储和并行计算完美地结合在一起。

(6) Hive(数据仓库)是基于Hadoop的一个数据仓库工具,可以将结构化的数据文件映射为一张数据库表,通过类SQL语句快速实现简单的MapReduce统计,不必开发专门的MapReduce应用,十分适合数据仓库的统计分析。

(7) Pig(一种强大的脚本语言)是一个基于Hadoop的大规模数据分析工具,它提供的SQL-LIKE语言叫Pig Latin,该语言的编译器会把类SQL的数据分析请求转换为一系列经过优化处理的MapReduce运算。

(8) HCatalog是基于Hadoop的数据表和存储管理服务,提供了更好的数据存储抽象和元数据服务。

(9) Sqoop(数据库同步工具)是一个用来将Hadoop和关系型数据库中的数据相互转移的工具,可以将一个关系型数据库如MySQL、Oracle、Postgres等中的数据导入到Hadoop的HDFS中,也可以将HDFS的数据导出到关系型数据库中。

(10) Flume(日志收集工具)是一个分布的、可靠的、高可用的海量日志聚合的系统,可用于日志数据收集、日志数据处理、日志数据传输。

(11) Chukwa(分布式数据采集系统)是一个开源的用于监控大型分布式系统的数据收集系统,是构建在Hadoop的HDFS和MapReduce框架之上的,继承了Hadoop的可伸缩性和鲁棒性。Chukwa还包含一个强大和灵活的工具集,可用于展示、监控和分析已收集的数据。

(12) Ambari(部署管理工具)是一种基于Web的工具,支持Apache Hadoop集群的供应、管理和监控。Ambari目前已支持大多数Hadoop组件,包括HDFS、MapReduce、Hive、Pig、HBase、ZooKeper、Sqoop和HCatalog等,也是5个顶级Hadoop管理工具之一。

(13) Oozie(作业流调度系统)是一个工作流引擎服务器,用于管理和协调运行在Hadoop平台上的HDFS、Map/Reduce和Pig任务工作流,同时Oozie还是一个Java Web程序,运行在Java Servlet容器中。

(14) ZooKeeper(分布式协调服务)是一个为分布式应用所设计的分布的、开源的协调服务,它主要是用来解决分布式应用中经常遇到的一些数据管理问题,简化分布式应用

协调及其管理的难度，提供高性能的分布式服务。

(15) Avro(数据序列化系统)可以将数据结构或者对象转换成便于存储和传输的格式，适合大规模数据的存储与交换。Avro 提供了丰富的数据结构类型、快速可压缩的二进制数据格式、存储持久性数据的文件集、远程调用 RPC 和简单动态语言集成等功能。

(16) Mahout(数据挖掘库)是基于 Hadoop 的机器学习和数据挖掘的一个分布式框架。Mahout 用 MapReduce 实现了聚类、分类、推荐引擎(协同过滤)和频繁集挖掘等广泛使用的数据挖掘方法。除了算法，Mahout 还包含数据的输入输出工具、与其他存储系统(如数据库、MongoDB 或 Cassandra)集成等数据挖掘支持架构。

3.3　Hadoop 生态系统

Hadoop 的各类组件功能各异，共同提供了互补性的服务，从而形成了一个海量数据处理的大数据生态系统。目前 Hadoop 生态系统根据 Hadoop 发行版本大体可以分为两类：Hadoop 1.0 生态系统和 Hadoop 2.0 生态系统。下面分别针对 Hadoop 1.0 生态系统和 Hadoop 2.0 生态系统的组成进行详细介绍。

3.3.1　Hadoop 1.0 生态系统

Hadoop 1.0 生态系统主要是指 Hadoop 1.×及其以前的版本(Hadoop 0.23.×除外)，并包含很多相关子系统的完整的大数据处理生态系统。图 3.3 展示了 Hadoop 1.0 生态系统的基础组成。

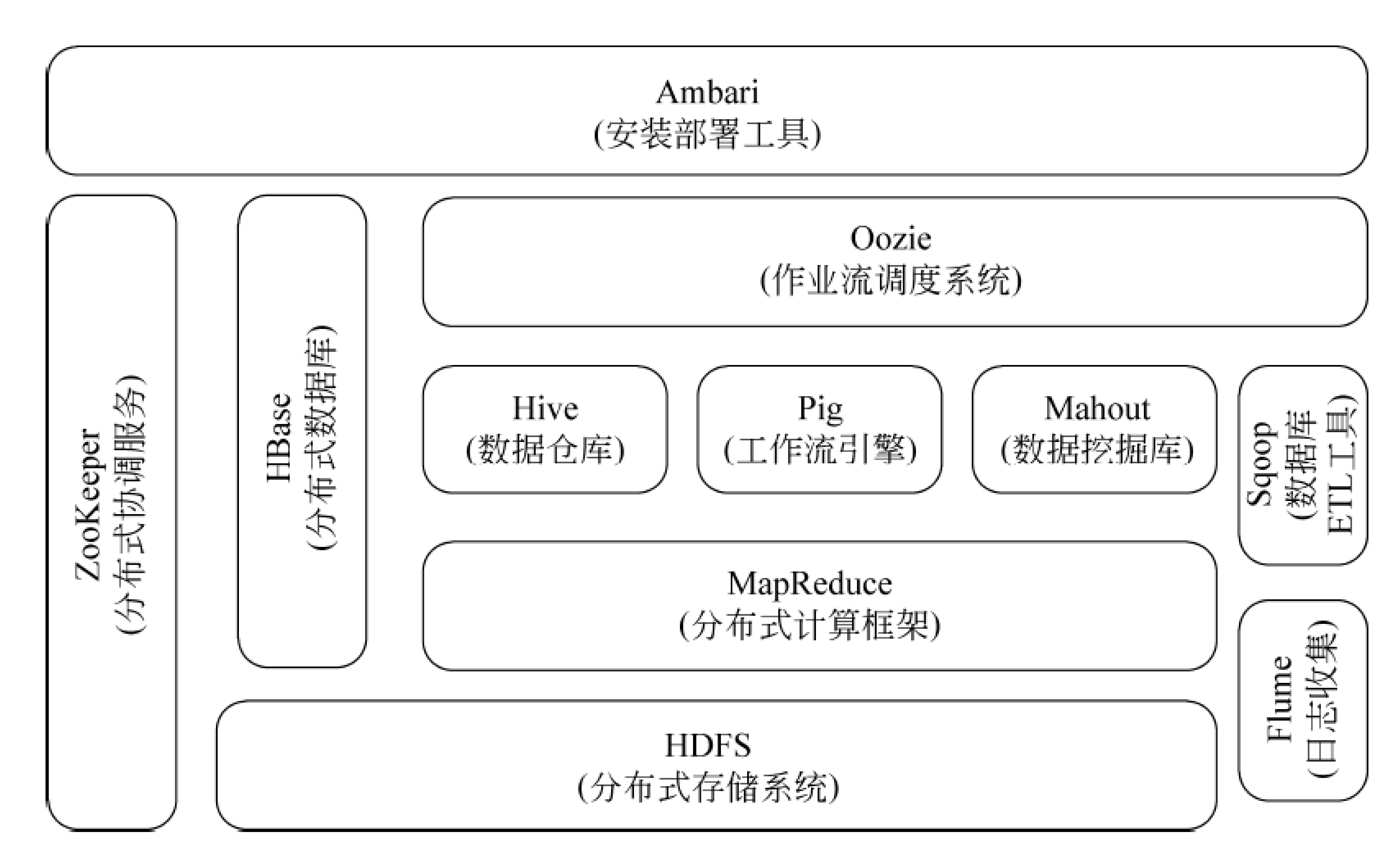


图 3.3　Hadoop 1.0 生态系统

从图 3.3 可以看出，Hadoop 1.0 生态系统在包含分布式文件系统 HDFS、分布式计算框架 MapReduce 和分布式数据库 HBase 等基本子系统的基础上，还包括如分布式协调服务 ZooKeeper、数据仓库 Hive、工作流引擎 Pig、数据挖掘库 Mahout、作业流调度系

统 Oozie、日志收集工具 Flume、数据库同步工具 Sqoop、安装部署工具 Ambari 等子系统，从而形成了 Hadoop 1.0 的生态系统。

3.3.2 Hadoop 2.0 生态系统

Hadoop 2.0 生态系统主要是指 Hadoop 2.X 及以后版本，并包含很多相关子系统的完整的大数据处理生态系统。图 3.4 展示了 Hadoop 2.0 生态系统的基础组成。

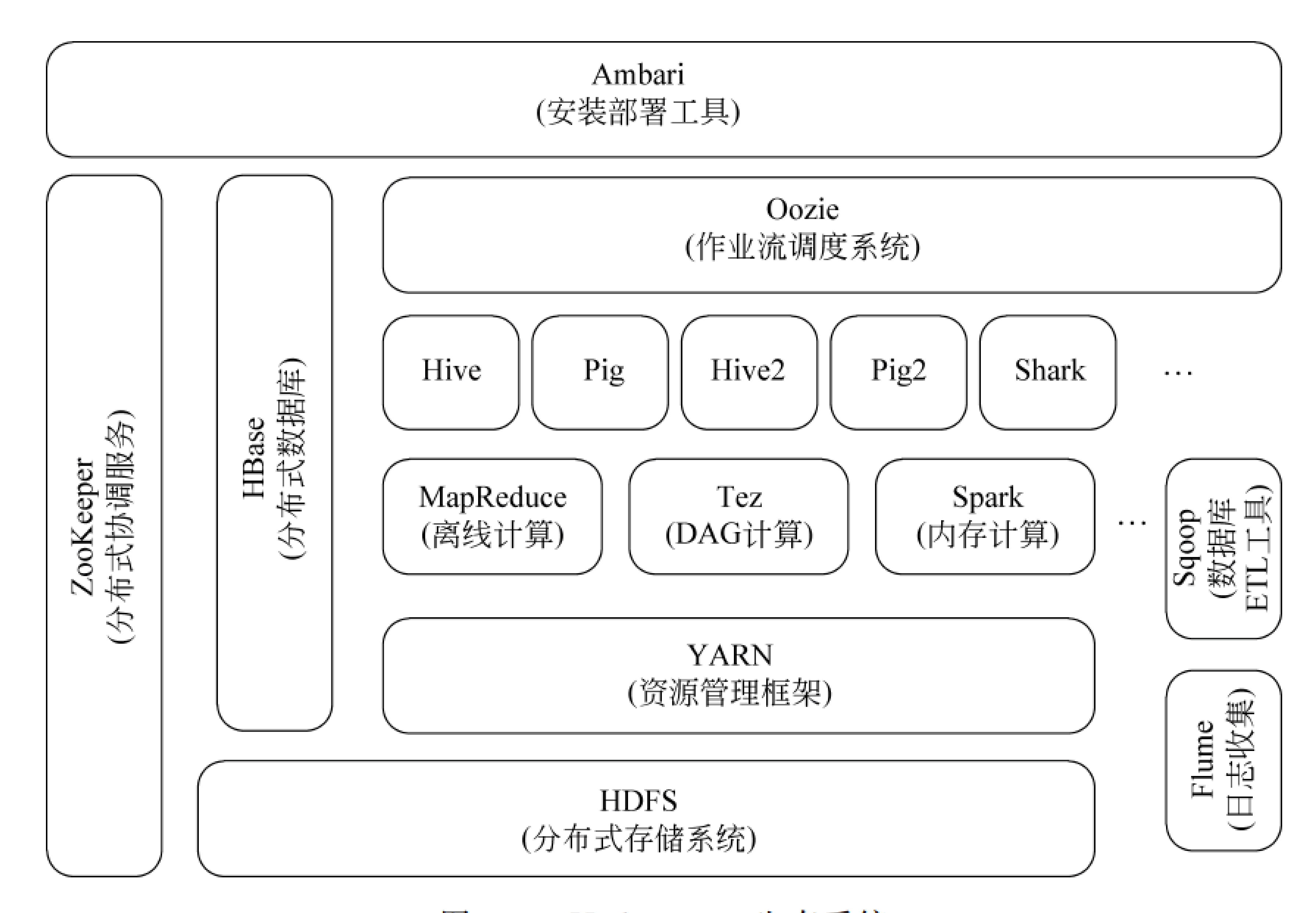


图 3.4 Hadoop 2.0 生态系统

从图 3.4 可以看出，与 Hadoop 1.0 生态系统不同之处在于采用了新的 MapReduce 框架(即 YARN，资源管理框架)原理，并增加了如 Spark、Tez 等子项目。资源管理在 Hadoop 生态系统中是很重要的一个模块，它直接决定了资源的组织形式和分配方式，是其他功能的基础。因此，Hadoop 2.0 生态系统针对 Hadoop 1.0 生态系统最大的优化和升级便是资源管理框架 YARN。

3.4 Hadoop 集群架构

Hadoop 集群使用了 Master/Slave 的架构模式，其集群架构主要由三部分组成：管理节点(MasterNode)、数据节点(SlaveNode)和客户端(Client)。管理节点主要负责管理集群节点，实现实体分配、负载均衡以及数据节点的失败恢复，并负责管理整个集群的 Meta 信息，并提供 Meta 信息服务；数据节点主要负责处理实际任务，如数据存储、子任务执行等，并通过心跳机制向管理节点定期汇报状态、工作进度等信息；客户端主要负责缓存集群以及 Meta 信息，避免与管理节点频繁通信，并且可以读写 API，进行批量操作等。



3.4.1　Hadoop 1.0 生态系统的集群架构

Hadoop 1.0 生态系统的集群架构主要是以 MapReduce 和 HDFS 为核心，其管理节点(MasterNode)主要负责两个核心功能：大数据存储(HDFS)和数据并行计算(MapReduce)的管理，如图 3.5 所示。

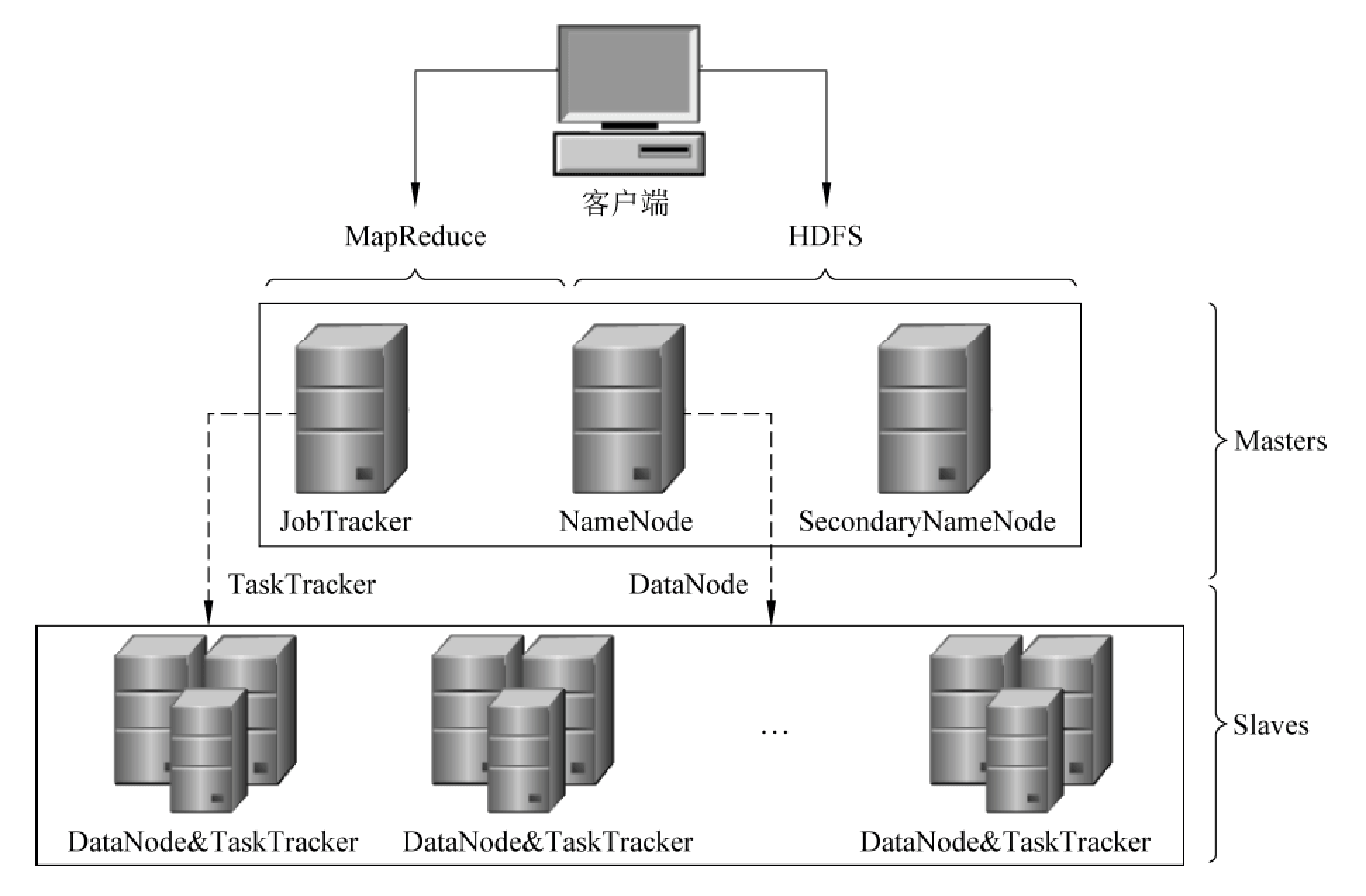


图 3.5　Hadoop 1.0 生态系统的集群架构

从图 3.5 可以看出，集群架构中的主要角色有 NameNode、JobTracker、DataNode 和 TaskTracker。其中，NameNode 负责监控和协调数据存储的工作，JobTracker 则负责 MapReduce 的并行计算。而数据节点(SlaveNode)则负责具体的工作以及数据存储。每个 Slave 运行一个 DataNode 和一个 TaskTracker 守护进程。这两个守护进程负责与管理节点(MasterNode)通信。其中，TaskTracker 守护进程与 JobTracker 相互作用，而 DataNode 守护进程则与 NameNode 相互作用。

注：对于较小的 Hadoop 集群(一般在 40 个节点左右)，可能存在一台服务器会扮演多个角色，如将 NameNode 与 JobTracker 部署在同一台服务器上，但对于大型的 Hadoop 集群(一般在 40 个节点以上)，为了保证集群的稳定性，应将 NameNode、SecondaryNameNode 和 JobTracker 三者分别部署于不同的服务器上。

3.4.2　Hadoop 2.0 生态系统的集群架构

Hadoop 2.0 生态系统的集群架构主要是以 MapReduce、HDFS 和 YARN 为核心，但总体上仍然是 Master/Slave 结构，如图 3.6 所示。

从图 3.6 可以看出，在整个集群中，YARN 为独立的资源管理与分配的通用系统，主

要由 ResourceManager、NodeManager 和 ApplicationMaster 和 Container 等几个组件构成。其中，ResourceManager 充当 Hadoop 1.0 生态系统中的 Master，NodeManager 充当 Hadoop 1.0 生态系统中的 Slave，ResourceManager 负责对各个 NodeManager 上的资源进行统一管理和调度。

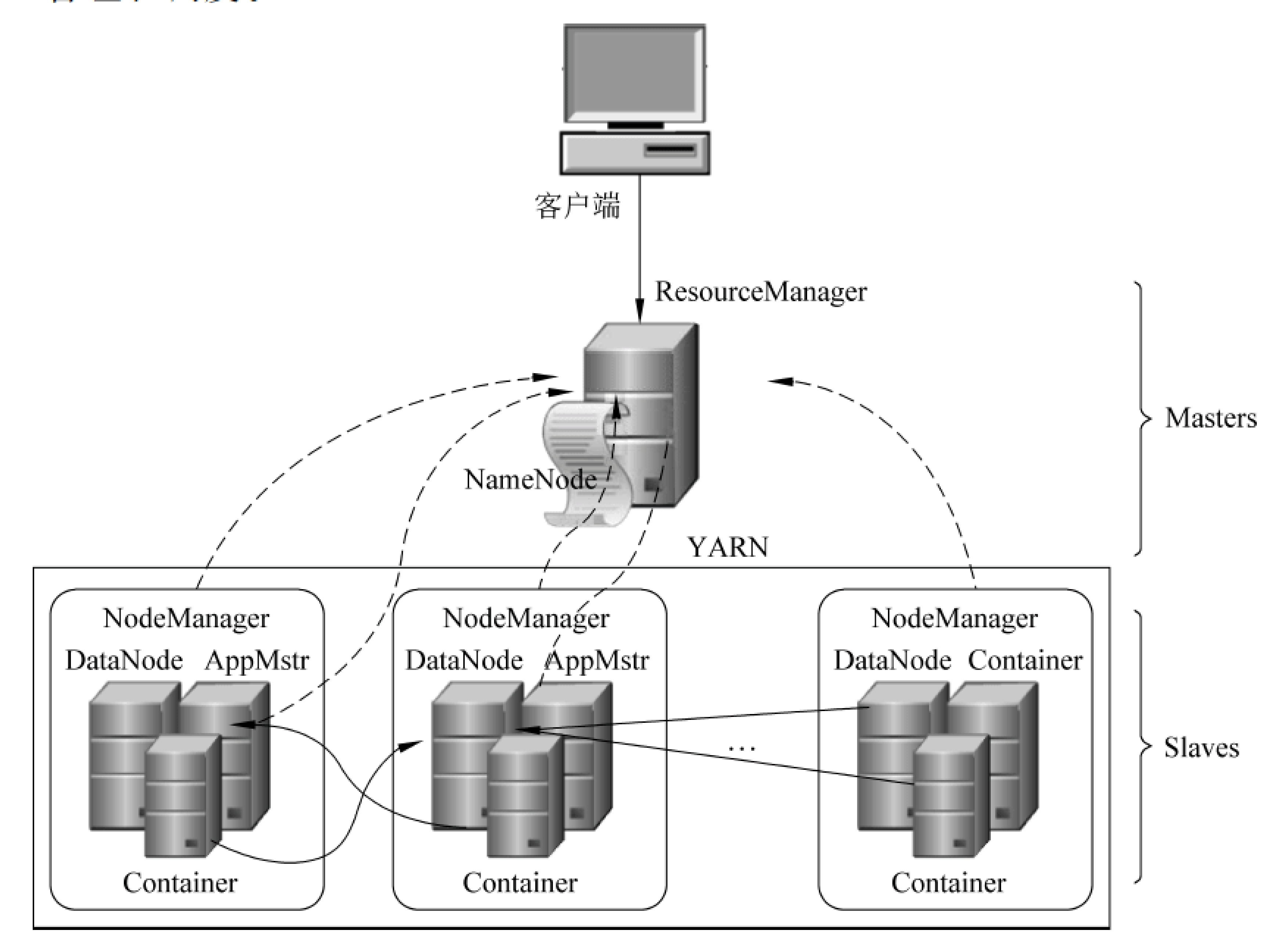


图 3.6 Hadoop 2.0 生态系统的集群架构

3.5 Hadoop 运行环境

要使得 Hadoop 集群能够充分发挥作用，就需要相应的软件、硬件及以太网络的支撑。下面将针对 Hadoop 系统正常运作所需要的硬件和软件环境以及 Hadoop 运行所需的网络环境进行介绍。

3.5.1 硬件环境

虽然 Hadoop 的一个优势是可以运行在普通的商用服务器上，但并不意味着 Hadoop 对硬件环境没有太高要求，而是要对自己所需要处理的问题有全面的了解，并根据 Hadoop 上面运行的应用程序的特性来确定其硬件环境，如对于机器学习、数据挖掘等计算密集型的应用则需要选用计算性能比较高的商用服务器；对于索引、检索、统计、聚类等 I/O 密集型的应用则需要选用 I/O 性能比较好的商用服务器等。因此，Hadoop 的硬件环境依据需求不同，其具体的硬件环境也会有较大差异。但是，要保证 Hadoop 运行的硬件环境满足最基本的稳定性和性能需求。

在图 3.5 给出的 Hadoop 1.0 生态系统的集群架构中，Master 节点、Slave 节点和客户端三者对硬件环境的要求并不完全相同。对于 Master 节点而言，一旦 Master 节点出



现故障将很可能导致集群所提供的服务中断；对于 Slave 节点而言，Slave 节点的崩溃属于正常现象，并不会对集群的可用性造成太大影响；对于客户端而言，客户端节点的故障将会影响作业的批量操作。因此，对于 Master 节点硬件需求的特点是高内存、低存储需求；Slave 节点既是存储也是计算，对硬件需求要考虑有足够的存储空间和足够的计算能力(CPU 速度和内存大小)；对客户端节点的硬件需求则要考虑其稳定性和满足应用需求。表 3.4 给出了 Hadoop 小型集群(40 节点以内)的各个节点的一些参考配置，供读者在硬件选型上作为参考基线。

表 3.4　Hadoop 集群各节点配置基线

服务器角色 及服务类型	功　能	配置基线	说　明
Master	运行 NameNode、JobTracker 和 SecondaryNameNode 节点	CPU：双四核英特处理器 内存：24GB DDR3 网卡：2×1GB Ethernet 硬盘：至少两块 SATA	(1) 如果中等规模集群(400 节点以上)，就要考虑内存翻倍，即考虑再增加 24GB 内存； (2) 对于大型集群，最好再翻倍，即 96GB 以上会更好一些
NameNode	记录 HDFS 中元数据，即包括文件名、权限、所有者、所有组、每个文件对应的 Block 列表，以及每个 Block 的副本目前存在于哪个机器上	内存：足够的内存 硬盘：适当的专用硬盘(稳定性至关重要)	(1) NameNode 信息会随着集群的使用以及规模而增加； (2) 大约一百万个 Block(64MB 或 128MB)或文件，会占据 NameNode 1GB 的内存
SecondaryNameNode	功能同 NameNode	内存：足够的内存 硬盘：专用硬盘(稳定性至关重要)	(1) 建议与 NameNode 使用一样的硬件配置，便于维护管理； (2) Hadoop 2.0 生态系统支持 NameNode HA，而其中的 StandbyNameNode 取代了 SecondaryNameNode，但两者功能相同，因此，二者硬件配置一样
JobTracker	在内存中记录所有 Job 和 Task 的状态、计数器、进行情况等	内存：满足需求大内存	(1) JobTracker 在内存中默认保留 100 个运行过的 Job 信息； (2) JobTracker 的内存使用情况是无法估计的，一定要关注 JobTracker 的内存占用情况
DataNode	每个节点同时既是计算也是存储	CPU：双六核英特处理器 内存：64GB DDR3 网卡：双 1GB 网卡 硬盘控制器：SAS 6GB/s 硬盘：12×3TB SATA 网络：2×1GB Ethernet	(1) 对于存储，由于 HDFS 默认是三个副本，如果系统每天产生 1TB 数据，则 HDFS 的需求就增长 3TB； (2) 对于计算，运行 MapReduce 需要一定的临时空间，一般考虑按照磁盘空间的 20%～30%作为 MapReduce 临时目录

注：对于所有集群中的服务器(除了 NameNode 和 SecondaryNameNode 节点外)，建议在物理硬盘中不要使用 RAID，如果 RAID 无法被移除，可将每个物理硬盘单独设置为 RAID 0。

3.5.2 软件环境

Hadoop 不仅需要硬件环境的支撑，同样也需要软件环境的支撑。其中，Hadoop 的软件环境主要包括支撑 Hadoop 运行的操作系统、Hadoop 运行环境和 Hadoop 节点之间的安全通信协议。

1. 操作系统

由于 Hadoop 是在 Linux 环境下开发的，一般来说会选择的操作系统也为 Linux 操作系统。任何一个支持 Java 1.6 的 Linux 操作系统都可以运行 Hadoop，如 Red Hat Enterprise Linux、CentOS、Ubuntu Server Edition、SuSE Enterprise Linux、Debian、Oracle Linux 等操作系统环境都与之匹配。但是，从 Hadoop 2.0 生态系统开始，已经支持在 Windows 操作系统上运行了，并且 HortonWorks 和微软公司合作，所开发的 HDP 的 Hadoop 版本是有 Windows 发布版本的。因此，操作系统的选择主要取决于该系统对硬件的支持能力、系统管理人员对系统的熟悉程度和对目前所使用的商业软件的支持能力等。

目前，Hadoop 系统仍大多运行在 Linux 系统上，并已在有 2000 个节点的 GNU/Linux 主机组成的集群系统上得到验证。在 Windows 环境中安装的 Hadoop，是作为开发平台支持的，但由于分布式操作尚未在 Windows 平台上充分测试，所以只是以学习和研究为目的，暂时不建议作为一个生产平台使用。

2. Java 运行环境

Hadoop 系统本身是用 Java 语言编写的，但也有少量的 C/C++ 代码。因此，Hadoop 的正常运行需要 JDK(Java Development Kit)的支持。在安装 JDK 时，也不建议只安装 JRE(Java Runtime Environment)，建议直接安装 JDK。因为 Hadoop 中的 MapReduce 程序的编写和 Hadoop 的编译都需要使用到 JDK 中的编译工具，而 JRE 无法满足需求，并且安装 JDK 时，可以同时安装 JRE。然而 JDK 的版本繁多，具体的兼容性情况请查看表 3.5 所给出的产商官方及一些用户的兼容性测试通过的 JDK 版本。

表 3.5 Hadoop 与 JDK 兼容性测试通过的版本

(数据来源：http://wiki.apache.org/hadoop/HadoopJavaVersions)

JDK 版本	测试产商
Oracle 1.6.0_16	Cloudera
Oracle 1.6.0_18	不详
Oracle 1.6.0_19	不详

续表

JDK 版本	测 试 产 商
Oracle 1.6.0_20	LinkedIn、Cloudera
Oracle 1.6.0_21	Yahoo!、Cloudera
Oracle 1.6.0_24	Cloudera
Oracle 1.6.0_26	Hortonworks、Cloudera
Oracle 1.6.0_28	LinkedIn
Oracle 1.6.0_31	Cloudera、Hortonworks
Oracle 1.7.0_15	Cloudera
Oracle 1.7.0_21	Hortonworks
Oracle 1.7.0_45	Pivotal
Openjdk 1.7.0_09-icedtea	Hortonworks

从表 3.5 可以看出，使用 Oracle Sun 的标准的 JDK 1.6.×环境，这是一个通过测试的环境，推荐使用 JDK 1.6 以后的版本；如果是 Java 7，则可以使用系统默认的 Openjdk 1.7。其中，Hortonworks 公司已经验证 JDK 1.6.0_31 在 RHEL 5/CentOS 5，RHEL 6/CentOS 6 和 SLES 11 操作系统下对 Hadoop 1.×，HBase，Pig，Hive，HCatalog，Oozie，Sqoop 和 Ambari 具有很好的兼容性；JDK 1.7.0.21 在 RHEL 5/CentOS 5，RHEL 6/CentOS 6 和 SLES 11 操作系统下对 Hadoop 2.2.0，HBase 0.96，Pig，Hive，HCatalog，Oozie，Sqoop 和 Ambari 具有很好的兼容性；Openjdk 1.7.0_09-icedtea 在 RHEL 6 操作系统下对 Hadoop 2.2.0，HBase 0.96，Pig，Hive，HCatalog，Oozie，Sqoop 和 Ambari 具有很好的兼容性。

3. 安全通信协议 SSH

Hadoop 是一个集群的环境，集群中的管理节点（MasterNode）需要对集群中的其他节点的服务进程进行远程的启动和停止时，就需要使用 SSH 协议。SSH 工具能够用来启动远程的命令，通过 SSH 工具能够在一个中心的管理节点上远程启动集群中的其他节点的服务进程，如 NameNode 节点使用 SSH 无密码登录并启动 DataNode 进程，同样，DataNode 上也能使用 SSH 无密码登录到 NameNode 节点。

1）NameNode 无密码登录所有 DataNode

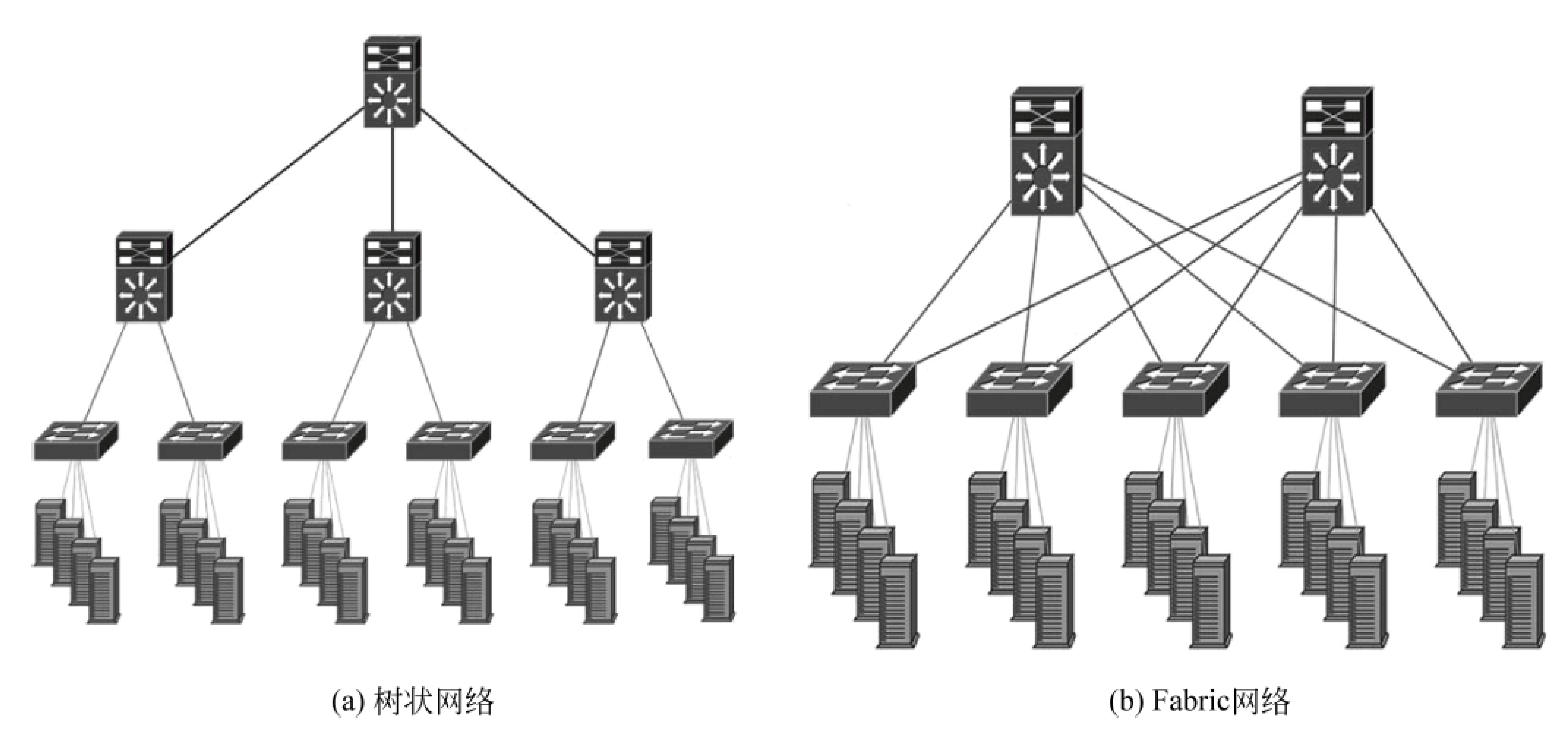
将 NameNode 作为客户端，生成一个密钥对，包括一个公钥和一个私钥，然后将公钥复制到 DataNode 上。当 NameNode 通过 SSH 连接 DataNode 时，DataNode 就会生成一个随机数并用 NameNode 的公钥对随机数进行加密，并发送给 NameNode。NameNode 收到加密数之后再用私钥进行解密，并将解密数回传给 DataNode，DataNode 确认解密数无误之后就允许 NameNode 进行连接。此过程中，不需要用户手工输入密码，只需要将 NameNode 上的公钥复制到 DataNode 上。

2）DataNode 无密码登录 NameNode

和 NameNode 无密码登录 DataNode 原理相同，只需要把 DataNode 的公钥复制到 NameNode 上。整个过程只涉及创建密钥、复制公钥、添加公钥内容，并不需要更改配置文件。

3.5.3 网络环境

由于 Hadoop 中的 MapReduce 在执行作业调度时，需要进行 Map 和 Reduce 两个过程，虽然 Hadoop 在 Map 阶段进行任务调度时，会尽量使任务本地化，但是对于 Reduce 过程仍会产生大量的 I/O。因此，Hadoop 集群网络在任意节点间的带宽需求都很高。如果网络拓扑结构设计时采用层级很多的树状网络，就会降低网络性能，如图 3.7(a)所示；如果网络拓扑结构设计时采用层级很少的 Fabric 网络，将会提高集群的网络性能，如图 3.7(b)所示。



(a) 树状网络　　(b) Fabric网络

图 3.7　网络拓扑结构

另外，为了使得 Hadoop 集群能够正常运行，建议最低使用千兆以太网连接，更高的带宽会带来更好的性能。由于网络内部有数据交换的需求，建议配置大容量的网络交换机，可以重点关注交换机的背板带宽。具有良好背板交换能力的交换机能使得任意两个接口之间的速度以及上行速度都能达到千兆的速度，而通过总线进行共享的带宽很难达到理想的速度。如果整个集群的负载比较高或者需要通过绑定两个以上 1GB 网卡来增加带宽时，就需要考虑部署 10GB 以太网络；如果需要的通信量更大或者需要混合一些高性能计算程序，就需要更加高端的网络配置，推荐使用 InfiniBand 网络，该网络具有更好的实现机制，能够减少网络的冲突。

3.6　Hadoop 集群的安装与配置

3.5 节已经介绍进行 Hadoop 集群的安装配置至少需要的软硬件环境和网络环境，本节将通过虚拟化技术来构建满足 Hadoop 部署的软硬件环境和网络环境。另外，本节



中的软件环境采用了 JDK 1.7 版本,VMware Workstation 虚拟机,CentOS 7 操作系统,针对 Hadoop 1.0 生态系统选用的 Hadoop 1.2.1 stable 版本,针对 Hadoop 2.0 生态系统选用的 Hadoop 2.6.0 stable 版本,并将重点介绍如何在 Linux 下快速搭建 Hadoop 环境,帮助初学者对 Hadoop 系统进行更深入的研究和学习。

3.6.1 准备工作

一个完整的 Hadoop 集群的安装,首先要做好 Hadoop 集群的规划,即要确保用于构建集群的所有服务器满足集群节点要求(软件环境要求、硬件环境要求和网络环境要求),然后在集群中的所有节点上安装相应的操作系统、配置 Hadoop 系统的软件环境、安装 Hadoop、进行集群参数配置等一系列工作。因此,在进行 Hadoop 集群安装之前,首先要进行集群规划和环境准备这两个过程。

1. 集群规划

集群规划主要是根据业务需求规划所使用的 Hadoop 组件,规划集群的硬件参数,规划集群使用的网络,规划各节点的 IP 地址及节点的角色等。例如,对 Hadoop 的组件规划包括可能会使用到的 HDFS、MapReduce、Hive、HBase 等;对集群的硬件参数规划包括服务器数量、物理布局、机架数据以及服务器在机架上的分配等;对集群所使用的网络规划包括集群网络的拓扑结构、节点到交换机的连接、机柜之间的连接等;对各节点 IP 地址及节点角色的规划包括各节点 IP 地址的分配、确定 Hadoop 的各个角色运行在哪些节点上等。因此,针对不同规模的 Hadoop 集群有不同的集群规划方案。表 3.6 给出了一个 Hadoop 1.0 生态系统集群的典型节点分配方案,读者可根据实际业务需求情况进行相应的调整。

表 3.6 Hadoop 集群典型的节点分配

角色	描述	节点数目
NameNode	分布式文件系统用以存储文件系统以及数据块的元数据	一个独立节点
SecondaryNameNode	NameNode 的备份节点	小规模集群可以和 NameNode 共享节点,大规模集群建议用独立节点
DataNode	HDFS 数据存储	多个独立节点
JobTracker	MapReduce 调试程序	一个独立节点,小规模集群可以与 NameNode 共享,大规模集群建议使用独立节点
TaskTracker	MapReduce 计算节点	与 DataNode 运行在相同的节点之上
Hive	Hive 元数据及驱动程序	独立配置的话可以与 NameNode 共享节点,或者将元数据存放在客户端
ZooKeeper	提供集群高可用性的锁服务	三个或三个以上的奇数据独立节点,小规模可以和其他角色共享节点
HBase HMaster	提供调试 RegionServer 的主模块	与其他角色共享节点的多个节点

续表

角　色	描　述	节点数目
HBase RegionServer	提供管理数据的模块	一般与 DataNode 运行在相同的节点上
ManagementNode	集群监控管理节点	一般为一个独立的节点，如果小规模集群可以与其他角色共享

在没有实际业务需求，以研究和学习为目的，并帮助初学者快速搭建 Hadoop 集群环境的情况下，我们使用最新的稳定版本 Hadoop 2.6.0（Hadoop 2.0 生态系统），并规划一个 Master 节点和三个 Slave 节点来搭建 Hadoop 集群。其中，Master 节点和 Slave 节点可使用 4 台装有 Linux 操作系统的普通 PC，也可由虚拟机创建（本实验环境是用 VMware 虚拟机创建的一个 Master 节点和三个 Slave 节点，其操作系统均为 CentOS 7），具体节点角色分配和配置参数信息如表 3.7 所示。

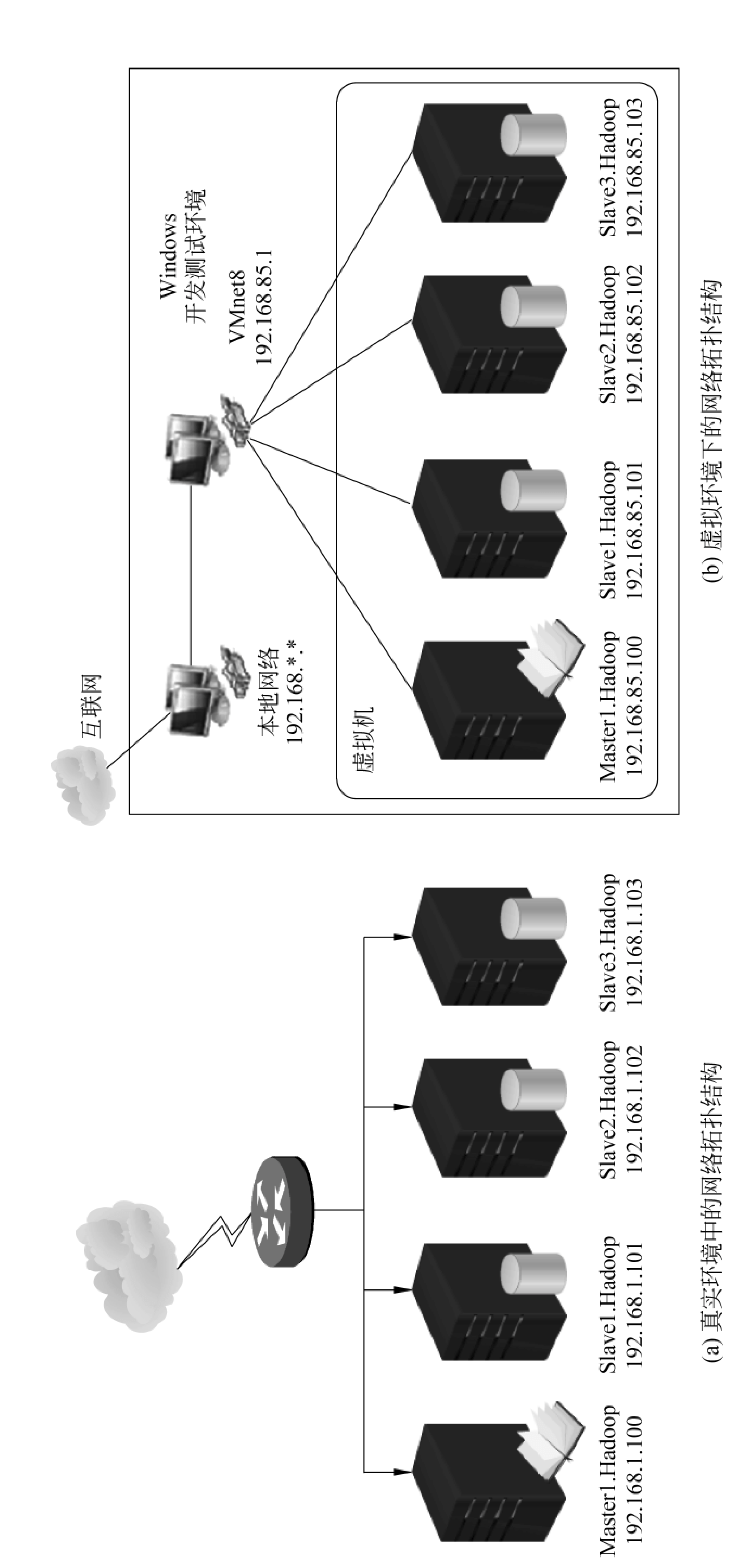
表 3.7　Hadoop 2.0 生态系统集群节点角色分配

机器名称	角　色	IP 地址	硬件参数	操作系统
Master1.Hadoop	NameNode SecondaryNameNode ResourceManager	192.168.1.100	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave1.Hadoop	DataNode NodeManager	192.168.1.101	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave2.Hadoop	DataNode NodeManager	192.168.1.102	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave3.Hadoop	DataNode NodeManager	192.168.1.103	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7

注： 在实际 Hadoop 集群环境中，由于 NameNode 对内存开销非常大，所以最好不要将 NameNode 和 SecondaryNameNode 部署在同一台服务器上。

由表 3.7 可以看出，Master 机器主要配置 NameNode 和 ResourceManager 的角色，负责总管分布式数据和分解任务的执行；Slave 机器配置 DataNode 和 NodeManager 的角色，负责分布式数据存储以及任务的执行；集群中各节点 IP 地址要设置在同一网段中（真实环境中使用的是 192.168.1.* 网段，虚拟机环境中使用的是 192.168.85.* 网段）；集群节点的操作系统均为 CentOS 7 操作系统。图 3.8 给出了该 Hadoop 集群网络的拓扑结构。其中，图 3.8(a)是在真实的环境中（一台路由连接的 4 台 PC）的网络拓扑；图 3.8(b)是使用 VMware 虚拟机环境下（用虚拟机创建了 4 台逻辑上独立存在的 PC）的网络拓扑结构。本质上图 3.8(a)和图 3.8(b)是同一个网络拓扑结构。

简单的 Hadoop 2.0 生态系统集群到此就规划完毕，下面将介绍对集群中的各个节点进行环境准备、Hadoop 安装和集群配置等内容。



(a) 真实环境中的网络拓扑结构

(b) 虚拟环境下的网络拓扑结构

图 3.8 Hadoop 集群网络拓扑结构

2. 环境准备

Hadoop 的安装部署需要相应的硬件和软件的支撑，而且分布式的存储和计算必然需要用到很多机器，在 Hadoop 部署时用户需要多台服务器来满足硬件需求。对大部分读者来说并没有直接适应 Hadoop 部署的硬件环境。因此，可以通过虚拟化技术来构建满足 Hadoop 部署的硬件环境，这就需要如 VMware Workstation、VirtualBox 等虚拟机工具的支持。具体 Hadoop 安装部署的准备环境过程如下。

1）安装 VMware 虚拟机

对于硬件条件有限的读者(主要针对 Windows 系统用户，对于 Linux 系统用户请忽略此步骤)，可以利用虚拟机来学习 Hadoop 的软硬件环境的配置工作。通过虚拟机可以将一台机器虚拟化成两台或多台机器，并且虚拟后的机器和实体机器使用上无任何区别，用户可把虚拟机当作实体机器来使用，并且在虚拟机上的误操作也不会对真实计算机造成任何影响。笔者使用的是 Windows 8 系统，在 Windows 系统下安装了 VMware Workstation 10 虚拟机。读者可在 VMware 公司官网(https://www.vmware.com/cn/)下载虚拟机或者查看本书的配套资料的软件目录中查找该软件。VMware 虚拟机的安装十分简单，在这里就不多赘述了，如有疑问可查阅相关资料。

2）VMware 引导 Linux 系统

本实例选用的 Linux 操作系统为 CentOS 7 版本，因为 CentOS 是目前主要应用于生产环境的 Linux 系统之一。读者可在 CentOS 官网(http://www.centos.org/)下载或者查看本书的配套资料的软件目录中查找该系统。安装过程如下。

注：*对于没有使用虚拟机的读者可以直接忽略上述过程，并直接阅读下面的 Liunx 操作系统安装步骤。*

双击 VMware Workstation 10 桌面图标，进入如图 3.9 所示的界面。



图 3.9　VMware Workstation 10 启动界面



单击“创建新的虚拟机”，进入选择安装类型，其中有两种安装类型：典型和自定义。在这里选择“典型”安装类型，如图 3.10 所示。



图 3.10　安装类型选择界面

单击“下一步”按钮，进入“新建虚拟机向导”，其中安装来源有三个选项：安装程序光盘、安装程序光盘映像文件和稍后安装操作系统。由于我们已从 CentOS 官网下载了 CentOS 7 的 Linux 操作系统，在这里选择“安装程序光盘映像文件”，并单击“浏览”按钮，选择 CentOS 7 的映像文件位置，如图 3.11 所示。



图 3.11　新建虚拟机向导

单击“下一步”按钮，选择虚拟机中要安装哪种类型的操作系统，并给出了相应的选项：Microsoft Windows、Linux、Novell NetWare、Solaris、VMware ESX 和其他。由于 CentOS 属于 Linux 操作系统，在这里选择 Linux，并选择 Linux 的版本为“CentOS 64 位”（这要根据所下载的 CentOS 版本来决定），如图 3.12 所示。

图 3.12 选择客户机操作系统

单击“下一步”按钮，进入命名虚拟机界面，可根据之前集群规划方案进行相应的命名，这里取名为“Master1. Hadoop”（建议取名尽量见名知意，方便后期的识别），并单击“浏览”按钮，选择虚拟机安装位置，这里使用默认位置 D:\VMwareHadoop\Master1. Hadoop，如图 3.13 所示。

图 3.13 命名虚拟机

单击"下一步"按钮,进入指定硬盘容量界面,"最大硬盘大小"为 20GB,并选择"将虚拟磁盘拆分成多个文件"(这两个选项都为默认设置),如图 3.14 所示。

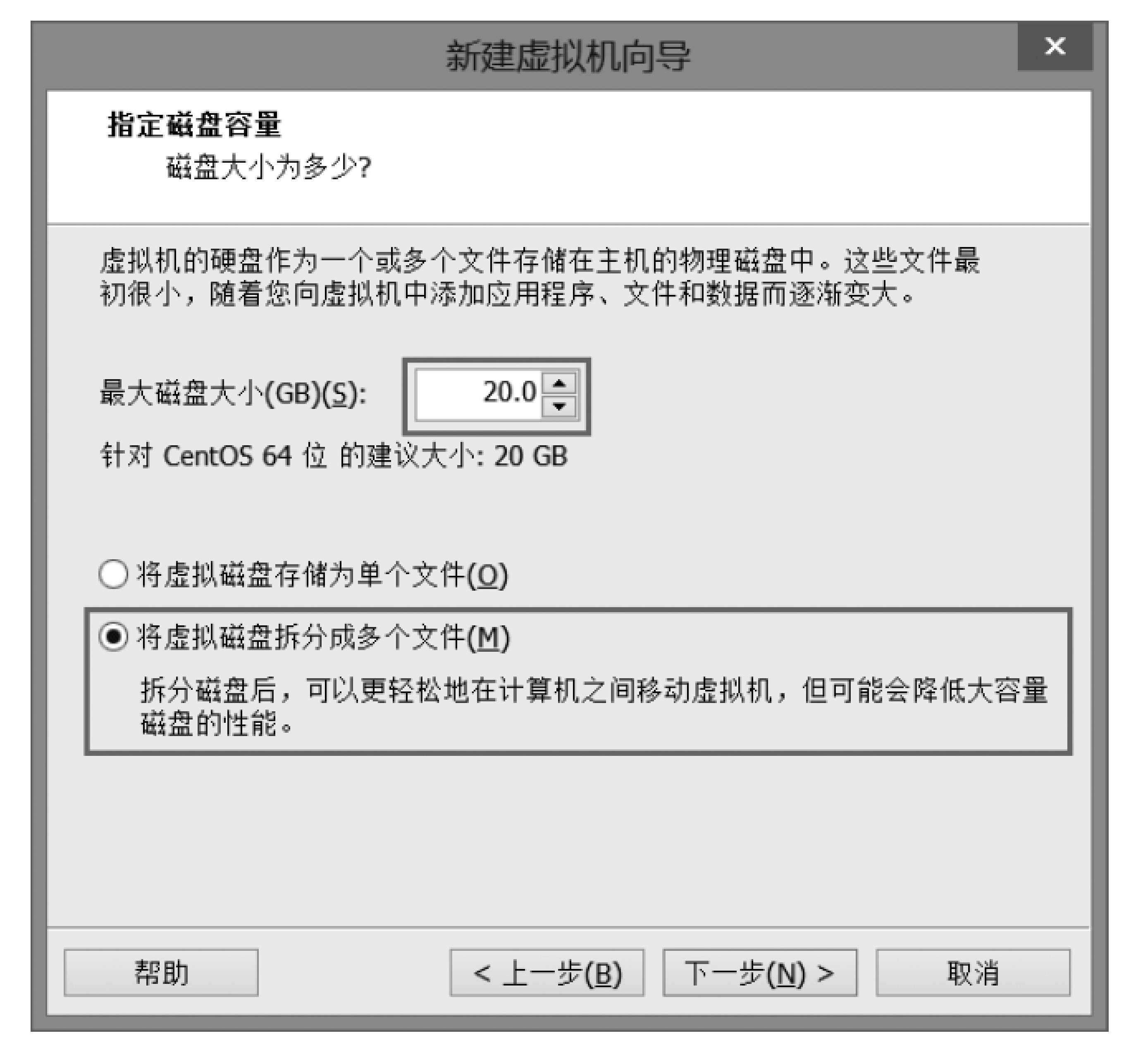


图 3.14 指定硬盘容量

单击"下一步"按钮,进入虚拟机的硬件环境选择界面,单击"自定义硬件"按钮,进行虚拟机的硬件环境选择。此 Master1. Hadoop 虚拟机配置参数均为默认设置即可,如图 3.15 所示。

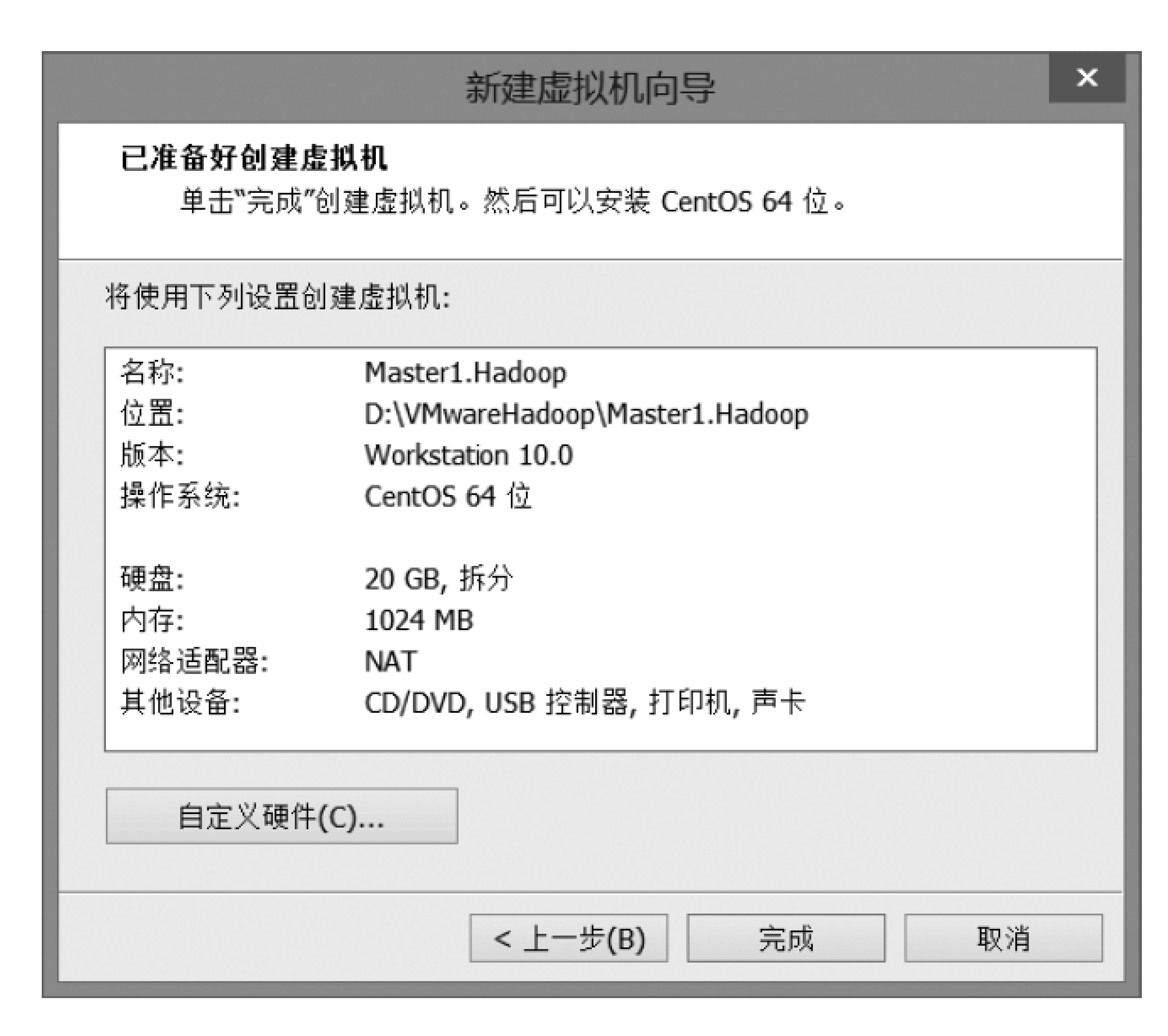


图 3.15 创建虚拟机的硬件环境

单击“完成”按钮，虚拟机创建完成，可以从 VMware Workstation 工作界面中看到虚拟机的名称“Master1. Hadoop”、虚拟机的硬件环境和有关虚拟机的详细信息，如图 3.16 所示。



图 3.16　虚拟机创建完成

最后，右击虚拟机名 Master. Hadoop→“电源”→“启动客户端”，完成虚拟机启动。

3）安装 Linux 操作系统

虚拟机启动后，将会引导 CentOS 系统的安装界面，选择 Install CentOS 7，如图 3.17 所示。



图 3.17　CentOS 系统安装

选择 Install CentOS 7 之后，将进行语言选择。对于测试环境而言，可以选择“中文”→“简体中文(中国)”；对于正式生产服务器建议选择英文版本。本实例选用英文版本，如图 3.18 所示。

单击 Continue 按钮，进入 CentOS 配置界面，有本地化、软件和系统三方面的配置，只有都配置完成后，Begin Installation 才可使用。这里可根据个人喜好进行选择，在安装



图 3.18　语言选择界面

源方面，使用的“本地介质”；在软件选择方面，选择了 GNOME Desktop；在安装位置方面，选择了 Automatic partitioning selected（自动分区）；在网络和主机名方面，选择了“开启网络”并命名主机名为“Master1. Hadoop”，IP 地址设置为静态 IP“192. 168. 85. 100”，网关和 NDS 都为“192. 168. 85. 2”，如图 3. 19 所示。



图 3.19　CentOS 7 系统配置界面

在网络和主机名方面，单击 NETWORK&HOSTNAME 进入网络信息界面，选择"开启网络"并命名主机名为"Master1. Hadoop"，如图 3.20 所示。

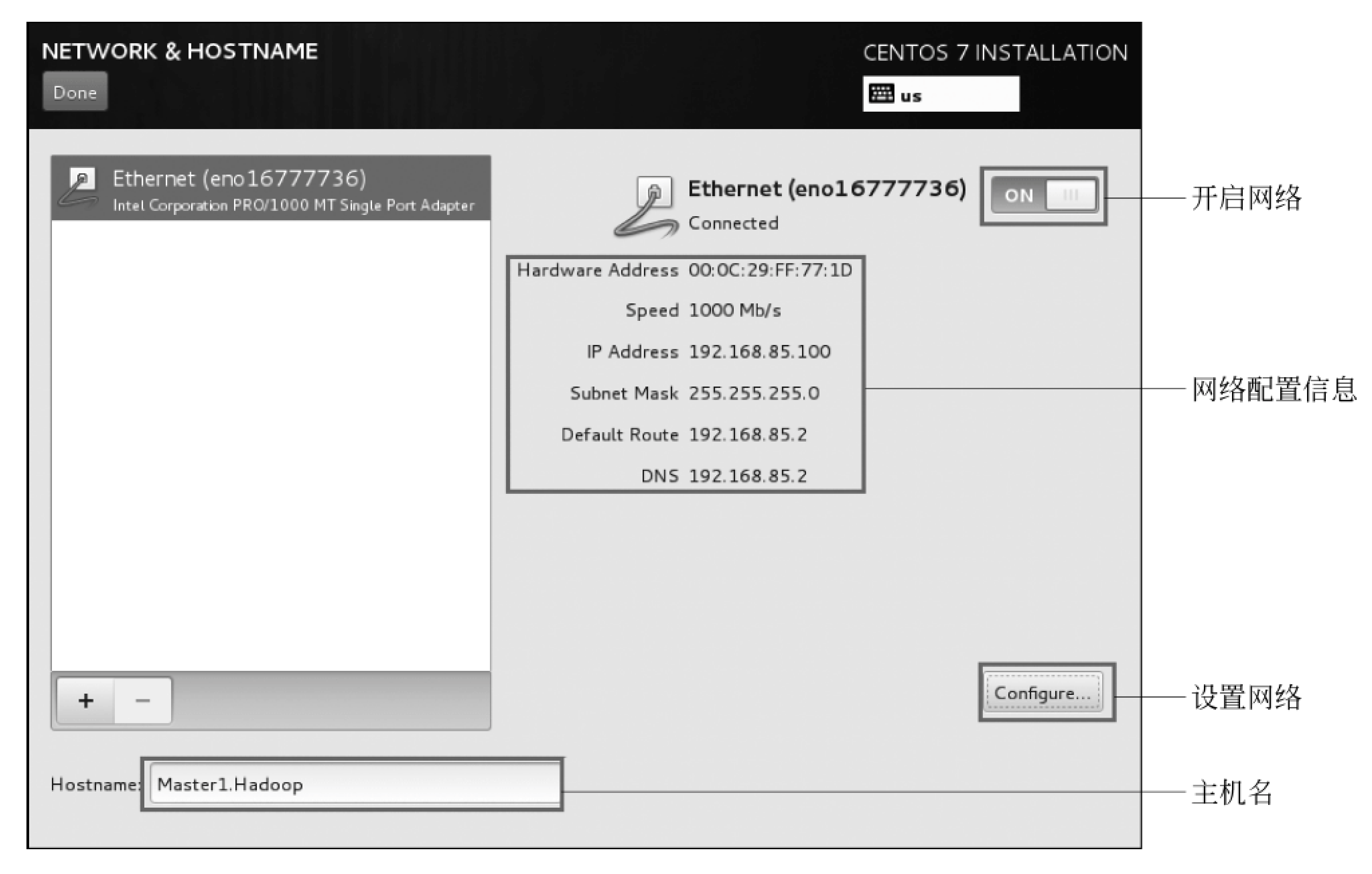


图 3.20　开启网络界面

单击 Configure 按钮进入网络设置界面，单击 IPv4 Settings，方法选择 Manual 手动设置模式，单击 Add 按钮将 IP 地址设置为静态 IP"192.168.85.100"，网关和 NDS 都为"192.168.85.2"。最后单击 Save 按钮完成网络设置，如图 3.21 所示。

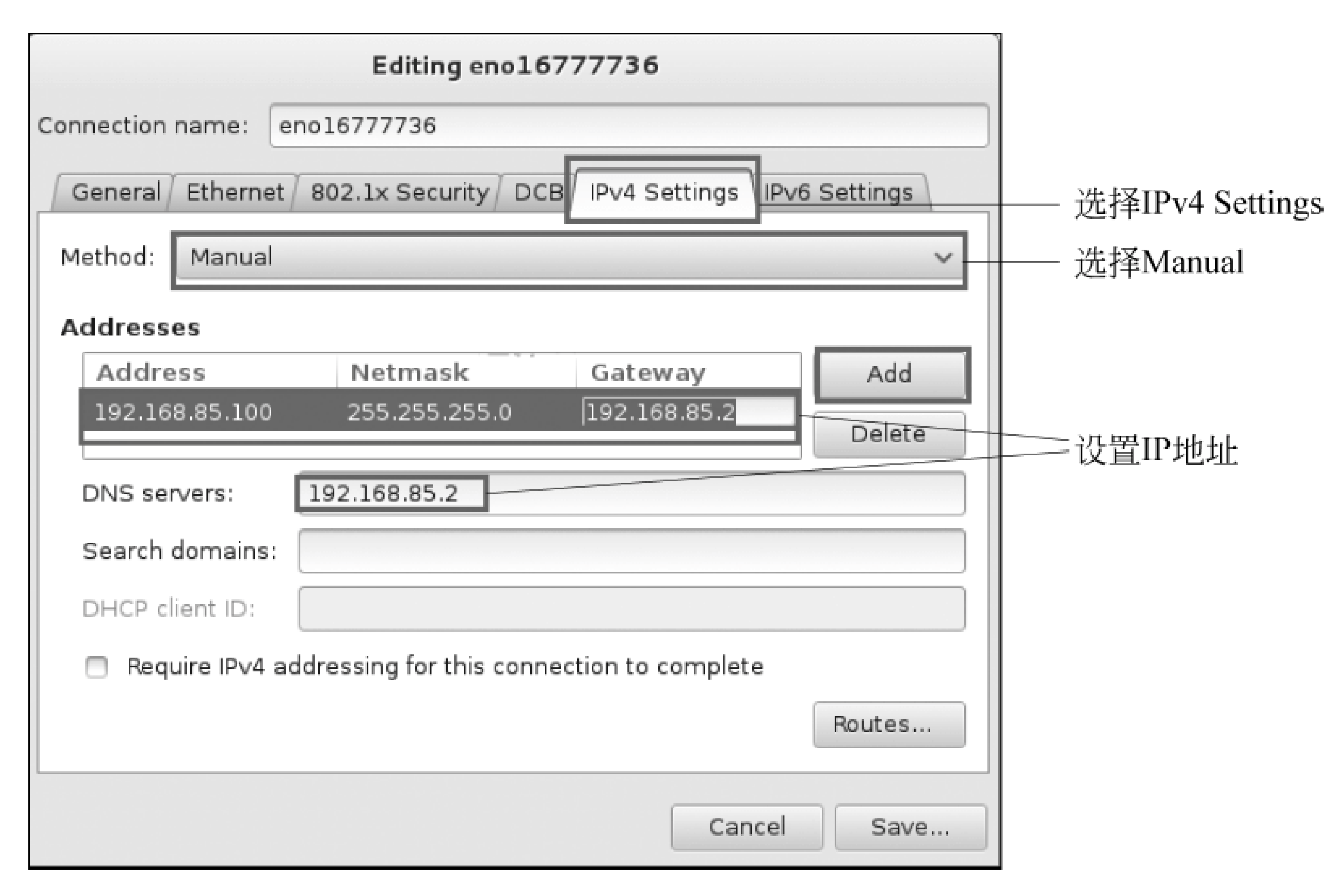


图 3.21　网络设置界面

单击 Begin Installation 按钮，进入安装界面并要求配置用户名和密码，在这里设置了 root 用户的密码为 root，新创建了用户名为 hadoop，密码为：hadoop 的新用户，如图 3.22 所示。

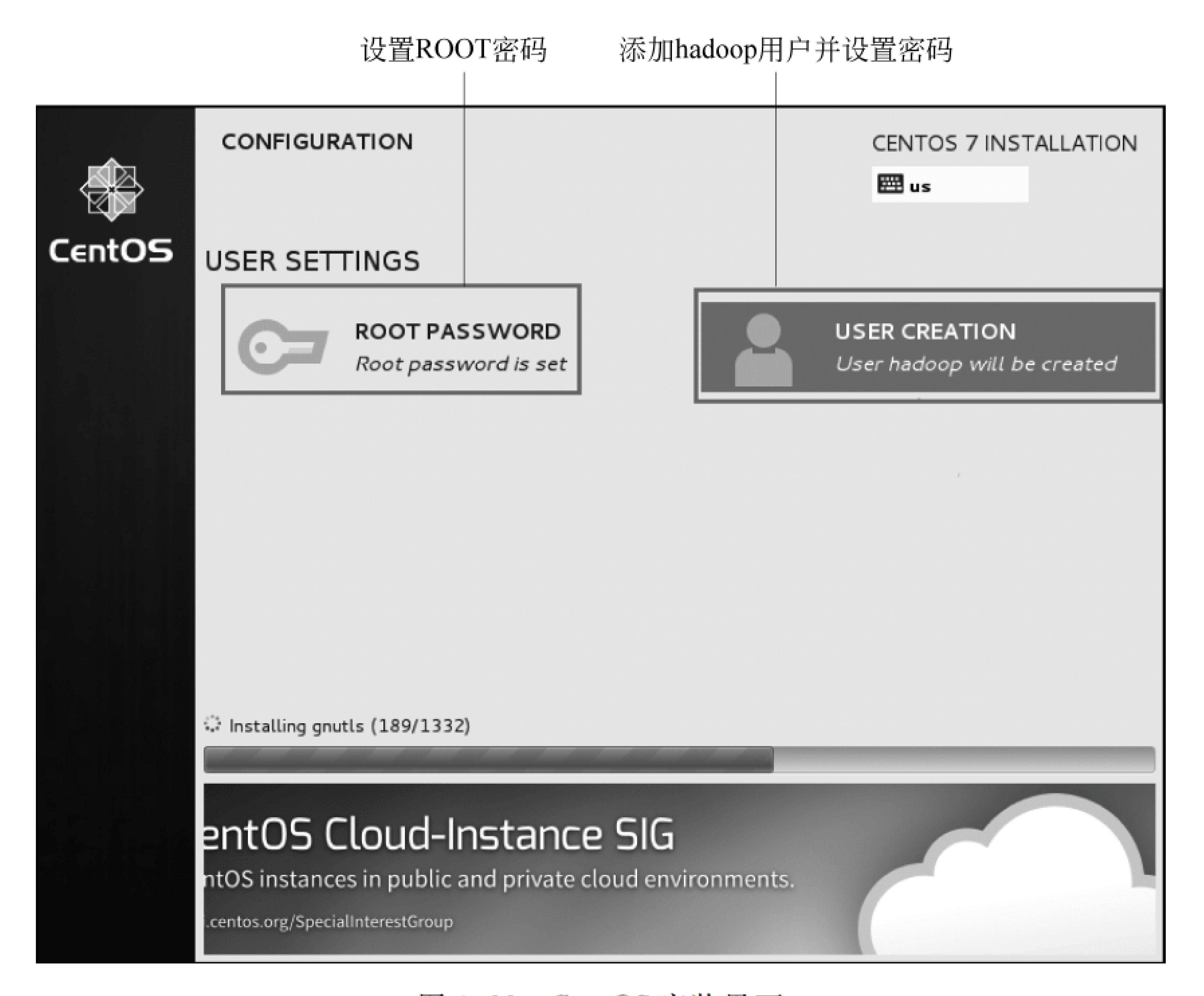


图 3.22　CentOS 安装界面

最后，安装完成后会提供重启，单击 Reboot，重启后进行"许可信息"、内核崩溃转储机制 Kdump 等配置，就完成了 CentOS 7 操作系统的安装。

4）网络配置

Hadoop 集群各个节点之间是互通的，这就需要对每个节点进行网络配置。网络配置可以在安装 CentOS 时进行配置（本实例是在安装 CentOS 7 系统时进行了网络配置，对于已进行网络配置的读者可以跳过此步骤），也可在 CentOS 安装完成之后，打开 CentOS 终端（Terminal）进行配置，具体的配置过程如下。

（1）关闭防火墙

首先要切换为 ROOT 用户，输入 ROOT 用户对应的密码（本实例 ROOT 用户对应的密码为：root）回车后，获得 ROOT 用户权限。查看防火墙是否处于开启状态，如果处于开启状态通过关闭防火墙命令将其停止运行，如图 3.23 所示。具体的操作命令行如下。

```
$ su - root                                   --切换 ROOT 用户
#systemctl status firewalld.service           --查看防火墙状态
#systemctl stop firewalld.service             --关闭防火墙
#systemctl disable firewalld.service          --永久关闭防火墙
```

```
[hadoop@localhost ~]$ su - root
Password:
Last login: Thu Dec 25 00:17:46 EST 2014 on pts/2
[root@localhost ~]# systemctl status firewalld.service
firewalld.service - firewalld - dynamic firewall daemon
   Loaded: loaded (/usr/lib/systemd/system/firewalld.service; enabled)
   Active: active (running) since Wed 2014-12-24 22:52:07 EST; 1h 27min ago
 Main PID: 760 (firewalld)
   CGroup: /system.slice/firewalld.service
           └760 /usr/bin/python -Es /usr/sbin/firewalld --nofork --nopid

Dec 24 22:52:07 localhost.localdomain systemd[1]: Started firewalld - dynamic...
Hint: Some lines were ellipsized, use -l to show in full.
[root@localhost ~]# systemctl stop firewalld.service
[root@localhost ~]# systemctl disable firewalld.service
rm '/etc/systemd/system/dbus-org.fedoraproject.FirewallD1.service'
rm '/etc/systemd/system/basic.target.wants/firewalld.service'
[root@localhost ~]# systemctl disable firewalld.service
[root@localhost ~]#
```


图 3.23　关闭防火墙

(2) 设置静态 IP 地址

如果在安装系统时没有配置好相应的 IP,就需要对/etc/sysconfig/network-scripts/ifcfg-eno16777736 文件进行配置,如图 3.24 所示。/etc/sysconfig/network-scripts/是文件目录,在该目录下存放的是网络接口的控制文件;ifcfg-eno16777736 是指当前机器的网络接口,修改完后可以通过"ifconfig"进行查看。具体的操作命令行如下。

```
#vim /etc/sysconfig/network-scripts/ifcfg-eno16777736
```

```
[root@localhost ~]# vim /etc/sysconfig/network-scripts/ifcfg-eno16777736
TYPE=Ethernet
BOOTPROTO=none
DEFROUTE=yes
IPV4_FAILURE_FATAL=no
IPV6INIT=yes
IPV6_AUTOCONF=yes
IPV6_DEFROUTE=yes
IPV6_FAILURE_FATAL=no
NAME=eno16777736
UUID=749a0069-77da-4eb0-a450-4e323c961bb5
ONBOOT=no
HWADDR=00:0C:29:FF:77:1D
IPADDR0=192.168.85.100
PREFIX0=24
GATEWAY0=192.168.85.2
DNS1=192.168.85.2
IPV6_PEERDNS=yes
IPV6_PEERROUTES=yes
```


图 3.24　设置当前机器 IP 地址

(3) 修改主机名

在本实例中的集群规划中,该节点的主机名为 Master1. Hadoop,对应的 IP 地址为 192.168.85.100。如果在安装系统时没有修改主机名,可通过命令行的方式修改主机名,如图 3.25 所示。在 CentOS 中,有三种定义的主机名:静态的(Static)、瞬态的(Transient)和灵活的(Pretty)。"静态"主机名是系统在启动时从/etc/hostname 自动初始化的主机名;"瞬态"主机名是在系统运行时临时分配的主机名,如通过 DHCP 或 DNS 服务器分配;"灵活"主机名则允许使用自由形式的主机名,以展示给终端用户。本实例所使用的主机名为静态主机名,具体的操作命令行如下。

```
#hostnamectl status                         --查看主机名相关的设置
#hostnamectl set-hostname Master1.Hadoop    --主机名设为:Master1.Hadoop
```

```
[root@localhost ~]# hostnamectl status      ←查看主机名
   Static hostname: localhost.localdomain   ↖静态主机名
         Icon name: computer
           Chassis: n/a
        Machine ID: 1a3ba578e6b843ed9a0c8a915823a121
           Boot ID: 831292bc1cf04de58cc099eb74a1b94b
    Virtualization: vmware
  Operating System: CentOS Linux 7 (Core)
       CPE OS Name: cpe:/o:centos:centos:7
            Kernel: Linux 3.10.0-123.el7.x86_64
      Architecture: x86_64
[root@localhost ~]# hostnamectl set-hostname Master1.Hadoop
[root@localhost ~]#    ↖同时修改静态、瞬态和灵活主机名
```

图 3.25　修改主机名

注：在修改静态/瞬态主机名时，任何特殊字符或空白字符会被移除，而提供的参数中的任何大写字母会自动转化为小写。一旦修改了静态主机名，/etc/hostname 将被自动更新。如果只想修改特定的主机名（静态、瞬态或灵活），可以使用“--static”、“--transient”或“--pretty”选项。

（4）配置 hosts 文件

/etc/hosts 文件是用来配置 DNS 服务器信息，即将主机名与 IP 地址一一对应起来。当用户在进行网络连接时，首先查找该文件，再寻找对应主机名的 IP 地址。如果没有配置该文件，可以 ping 通本机的 IP 地址，却无法 ping 通本机的主机名。具体的操作命令行如下。

```
#vim /etc/hosts                             --编辑 hosts 文件
--在 hosts 文件末尾添加集群中规划的 Master 和 Slave 主机及对应的 IP 地址
192.168.85.100 Master1.Hadoop
192.168.85.101 Slave1.Hadoop
192.168.85.102 Slave2.Hadoop
192.168.85.103 Slave3.Hadoop
```

这样 Master 与所有的 Slave 机器之间不仅可以通过 IP 进行通信，而且还可以通过主机名进行通信。现在可以对“Master1. Hadoop”主机名进行 ping 通测试，如果 hosts 文件配置成功，则可以测试成功，如图 3.26 所示。

```
[root@master1 ~]# ping Master1.Hadoop   ←测试ping主机名
PING Master1.Hadoop (192.168.85.100) 56(84) bytes of data.
64 bytes from Master1.Hadoop (192.168.85.100): icmp_seq=1 ttl=64 time=0.034 ms
64 bytes from Master1.Hadoop (192.168.85.100): icmp_seq=2 ttl=64 time=0.040 ms
64 bytes from Master1.Hadoop (192.168.85.100): icmp_seq=3 ttl=64 time=0.039 ms
64 bytes from Master1.Hadoop (192.168.85.100): icmp_seq=4 ttl=64 time=0.043 ms
^C
--- Master1.Hadoop ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3005ms
rtt min/avg/max/mdev = 0.034/0.039/0.043/0.003 ms   ↖表示配置成功
[root@master1 ~]#
```

图 3.26　配置 hosts 文件

(5) 创建 Hadoop 用户

在真实的集群分布模式下,要求集群中所有节点使用相同的用户名和密码,并且要求在所有节点上安装的 Hadoop 系统具有完全一致的目录结构。在本实例中的集群规划中,各节点的用户名和密码统一规划为:hadoop。如果在安装系统时没有创建 Hadoop 用户,可以通过命令行的方式进行添加,具体的操作命令行如下。

```
#useradd hadoop                    --创建用户名为 hadoop 的用户
#passwd hadoop                     --为用户 hadoop 设置密码
```

如果采用虚拟机的方式进行网络配置,VMware 提供了三种工作模式:桥接(bridge)、网络地址转换(NAT)和主机模式(host-only)。在桥接模式下,VMware 虚拟出来的操作系统就像是局域网中的一台独立主机,它可以访问网内任何一台机器,该模式下虚拟主机的 IP 地址要和主机 IP 地址在同一网段;在 NAT 模式下,虚拟系统的 TCP/IP 配置信息是由 VMnet8(NAT)虚拟网络的 DHCP 服务器提供的,无法进行手工修改,因此虚拟系统也就无法和本局域网中的其他真实主机进行通信,采用 NAT 模式最大的优势是虚拟系统接入互联网非常方便,不需要进行任何其他的配置,只需要宿主机器能访问互联网即可(本实例就是采用的 NAT 模式);在主机模式下,所有的虚拟系统可以相互通信,但虚拟系统和真实的网络是被隔离开的,相当于宿主机器和虚拟机是通过双绞线互连。

5) 安装 Java 运行环境

在 Hadoop 部署之前,集群上所有节点都需要安装 JDK 来支撑 Java 运行环境(JDK 版本的选择请参考表 3.5)。CentOS 7 已经自带了 OpenJDK 1.7 版本,如需要另安装 JDK,可在 Oracle 官网下载 JDK 的 rpm 包(此类包的 JDK 可直接在 Linux 系统中安装,而无须再进行其他配置),然后在 CentOS 7 的终端执行如下命令:

```
#yum - y remove java [jdk文件名]              --卸载 CentOS 7 自带的 OpenJDK1.7 版本
#rpm -ivh /[JDK位置]/[JDK文件名].rpm          --安装 JDK,默认安装在/usr/java/目录下
#java - version                               --验证是否安装成功
```

如果安装成功,将会显示当前 JDK 版本号,如图 3.27 所示。

```
[root@master1 hadoop]# java -version      ←检查是否安装成功
java version "1.7.0_71"                   ←JDK版本号
Java(TM) SE Runtime Environment (build 1.7.0_71-b14)
Java HotSpot(TM) 64-Bit Server VM (build 24.71-b01, mixed mode)
[root@master1 hadoop]#
```

图 3.27 JDK 版本号显示

JDK 安装成功之后,下一步将要配置三个环境变量:PATH、CLASSPATH 和 JAVA_HOME,三个变量的设置如表 3.8 所示。

表 3.8　三个变量的设置

变 量 名	变　量　值
JAVA_HOME	指明 JDK 安装路径(默认路径为/usr/java/目录下),此路径下包括 lib,bin,jre 等文件夹,JAVA_HOME 设置为: JAVA_HOME=/usr/java/jdk1.7.0_71
PATH	使得系统可以在任何路径下识别 Java 命令,PATH 设置为: PATH=$PATH:$JAVA_HOME/bin
CLASSPATH	CLASSPATH 为 Java 加载类路径,只有类在 CLASSPATH 中,Java 命令才能识别,CLASSPATH 设置为: .:$JAVA_HOME/jre/lib/rt.jar:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar 注意:CLASSPATH 变量是以“.:”开始,表示当前路径

修改/etc/profile 文件,将 JAVA_HOME、PATH 和 CLASSPATH 添加到 profile 配置文件中,具体执行命令如下。

```
#vim /etc/profile
export JAVA_HOME=/usr/java/jdk1.7.0_71
export CLASSPATH=.:$JAVA_HOME/jre/lib/rt.jar:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$PATH:$JAVA_HOME/bin
#source /etc/profile                          --立即生效
#echo $PATH                                   --查看 PATH 变量值
```

到此,Java 运行环境就安装完毕。集群规划中的 Slave1.Hadoop、Slave2.Hadoop 和 Slave3.Hadoop 节点的系统配置方法同 Master1.Hadoop 节点的系统配置方法相同,在这里就不再赘述。

6)配置 SSH 无密码登录

默认 CentOS 已经安装了 OpenSSH(即使是最小化安装),这里就不介绍 OpenSSH 的安装了,可以在 CentOS 7 的终端执行如下命令查看 ssh 和 rsync 服务(rsync 是一个远程数据同步工具,可通过 LAN/WAN 快速同步多台主机间的文件)。

```
$rpm - qa | grep openssh                      --检查是否安装 openssh
$rpm - qa | grep rsync                        --查看 rsync
$systemlctl status sshd.service               --查看 ssh 服务状态
```

如果没有安装 ssh 和 rsync 服务,可以通过下面的命令进行安装。

```
yum install ssh                               --安装 SSH 协议
yum install rsync                             --安装 rsync
systemctl enable sshd.service                 --开机启动 sshd 服务
```

如果服务启动成功,请查看如图 3.28 所示的结果。

执行 ssh-keygen 命令生成密钥文件,输入“ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa”,

```
[hadoop@master1 ~]$ systemctl status sshd.service
sshd.service - OpenSSH server daemon
   Loaded: loaded (/usr/lib/systemd/system/sshd.service; enabled)
   Active: active (running) since Thu 2014-12-25 20:11:50 EST; 1h 54min ago
  Process: 1090 ExecStartPre=/usr/sbin/sshd-keygen (code=exited, status=0/SUCCESS)
 Main PID: 1097 (sshd)
   CGroup: /system.slice/sshd.service
           └1097 /usr/sbin/sshd -D
```

表示服务已启动

图 3.28　ssh 服务启动成功

其中-t、-P、-f 参数区分大小写，ssh-keygen 是生成密钥命令；-t 表示指定生成的密钥类型(dsa，rsa)；-P 表示提供的密语；-f 指定生成的密钥文件；～代表当前用户的文件夹，即为/home/用户名。执行此命令后，将会在“\home\用户名”路径下面生成.ssh 文件夹(可通过命令“ls -a /home/用户名”查看，本实例为“ls -a /home/hadoop”)。

在 Master1.Hadoop 节点终端执行的具体命令如下。

```
#su hadoop                                        --切换到 hadoop 用户
        --生成密码对,/home/hadoop/.ssh/id_rsa 和 /home/hadoop/.ssh/id_rsa.pub
$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
        --把 id_rsa.pub 追加到授权的 key 里面去
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
$ chmod 600 ~/.ssh/authorized_keys                --修改权限
$ su- root                                        --切换到 root 用户
#vim /etc/ssh/sshd_config                         --修改 ssh 配置文件
RSAAuthentication yes                             #启用 RSA 认证
PubkeyAuthentication yes                          #启用公钥私钥配对认证方式
AuthorizedKeysFile .ssh/authorized_keys           #公钥文件路径
#su hadoop                                        --切换到 hadoop 用户
        --把公钥复制所有的 Slave 机器上
$ scp ~/.ssh/id_rsa.pub hadoop@192.168.85.101:~/
$ scp ~/.ssh/id_rsa.pub hadoop@192.168.85.102:~/
$ scp ~/.ssh/id_rsa.pub hadoop@192.168.85.103:~/
```

生成的密钥信息如图 3.29 所示。

```
[hadoop@master1 ~]$ ssh-keygen -t rsa -P '' -f ~/.ssh/id_rsa
Generating public/private rsa key pair.
Created directory '/home/hadoop/.ssh'.
Your identification has been saved in /home/hadoop/.ssh/id_rsa.
Your public key has been saved in /home/hadoop/.ssh/id_rsa.pub.
The key fingerprint is:
89:84:39:f3:b4:db:aa:eb:cd:c5:41:99:72:38:32:e6 hadoop@master1.hadoop
The key's randomart image is:
+--[ RSA 2048]----+
|                 |
|     o . o       |
|    o * =        |
|   o O B .       |
|    E + S        |
|       + .       |
|      . +        |
|     o o         |
|   .+o+          |
+-----------------+
```

生成密钥命令

生成的密钥信息

图 3.29　生成密钥

在所有 Slave 节点终端执行的具体命令如下。

```
#su hadoop                                          --切换到 hadoop 用户
$mkdir ~/.ssh
$chmod 700 ~/.ssh
$cat ~/id_rsa.pub >> ~/.ssh/authorized_keys         --追加到授权文件"authorized_keys"
$chmod 600 ~/.ssh/authorized_keys                   --修改权限
$su                                                 --切换回 root 用户
#vim /etc/ssh/sshd_config                           --修改 ssh 配置文件
RSAAuthentication yes                               --启用 RSA 认证
PubkeyAuthentication yes                            --启用公钥私钥配对认证方式
AuthorizedKeysFile .ssh/authorized_keys             --公钥文件路径
```

再在 Master 节点终端重启 SSH 服务，并验证 Master1. Hadoop 到 Slave1. Hadoop、Slave2. Hadoop、Slave3. Hadoop 之间的 SSH 无密码登录：

```
#systemctl restart sshd.service                     --重启 SSH 服务
#su hadoop                                          --切换回 hadoop 用户
$ssh Slave1.Hadoop                                  --Slave1.Hadoop 的 SSH 无密码验证
$ssh Slave2.Hadoop                                  --Slave2.Hadoop 的 SSH 无密码验证
$ssh Slave3.Hadoop                                  --Slave3.Hadoop 的 SSH 无密码验证
```

这样就实现了从 Master1. Hadoop 到 Slave1. Hadoop、Slave2. Hadoop、Slave3. Hadoop 的无密码登录，登录成功，将出现如图 3.30 所示的结果。如果要配置所有 Slave 节点无密码登录 Master 节点，和 Master 无密码登录所有 Slave 节点原理一样，这里不再赘述。

```
[hadoop@master1 ~]$ ssh slave1.hadoop
Last login: Sun Dec 28 21:41:06 2014 from 192.168.85.1
[hadoop@slave1 ~]$ exit
logout
Connection to slave1.hadoop closed.
[hadoop@master1 ~]$ ssh slave2.hadoop
Last login: Sun Dec 28 21:36:21 2014 from master1.hadoop
[hadoop@Slave2 ~]$ exit
logout
Connection to slave2.hadoop closed.
[hadoop@master1 ~]$ ssh slave3.hadoop
Last login: Sun Dec 28 21:39:49 2014 from master1.hadoop
```

SSH无密码登录
登录成功
退出Slave1.Hadoop节点
Slave2.Hadoop节点
Slave3.Hadoop节点

图 3.30 SSH 无密码登录成功界面

注： 如果使用虚拟机方式，可在 Windows 系统下安装 SSH Secure Shell 软件，该软件可用来连接远程 Linux 服务器或者虚拟机中的 Linux 操作系统。其中，Secure Shell 相当于控制台，SSH Secure File Transfer Client 则可以方便快捷地在 Windows 和 Linux 操作系统之间传输文件。

3.6.2 Hadoop 部署

Hadoop 部署方式有三种：单机模式、伪分布模式和全分布模式。在单机模式下(在一台运行 Linux 或 Windows 下虚拟 Linux 的单机上安装运行 Hadoop 系统)，无须运行任何守护进程，所有程序都在单个 JVM 上执行，在该模式下测试和调试 MapReduce 程序较为方便，因此，这种模式适宜用在开发阶段；在伪分布模式下(在一台运行 Linux 或 Windows 下虚拟 Linux 的单机上)，用不同的 Java 进程模拟分布运行中的 NameNode、DataNode、JobTracker、TaskTracker 等各类节点，模拟一个小规模的集群；在全分布模式下(在一个真实的集群环境下安装运行 Hadoop 系统，集群的每个节点可以运行 Linux 或 Windows 下的虚拟 Linux)，Hadoop 守护进程运行在一个集群上。在单机和伪分布模式下编写完成的程序不需修改即可在真实的分布式 Hadoop 集群下运行，但通常需要修改配置。

本实例将采用全分布模式进行 Hadoop 部署，因为这种方式更有利于读者深入理解 Hadoop 体系构架。所有的机器上都需要安装 Hadoop，现在就先在 Master1. Hadoop 节点进行 Hadoop 安装，然后其他服务器按照步骤重复进行即可。安装配置 Hadoop 需要以"root"用户身份进行，具体安装配置过程如下。

1. 安装 Hadoop

在 Hadoop 官网(http://hadoop. apache. org/releases. html)下载相应版本的 Hadoop，下载完成后解压到 CentOS 系统的/opt/目录下并进行一些基础配置，具体命令如下。

```
#tar - zxvf hadoop-2.6.0                              --解压 Hadoop 安装文件
#mv hadoop-2.6.0 /opt/hadoop-2.6.0                    --移动 hadoop-2.6.0 到"/opt/"目录下
                                    --把"/opt/Hadoop-2.6.0"读权限分配给 hadoop 用户
#chown -R hadoop:hadoop /opt/hadoop-2.6.0
#vi /etc/profile                    --把 Hadoop 安装路径添加到配置文件 profile 中
export HADOOP_HOME=/opt/hadoop-2.6.0
export PATH=$PATH:$HADOOP_HOME/bin
#su hadoop                          --切换为 hadoop 用户
$cd /opt/hadoop-2.6.0
$mkdir -p dfs/name                  --在"/opt/Hadoop-2.6.0"目录下创建"/dfs/name"目录
$mkdir -p dfs/data                  --在"/opt/Hadoop-2.6.0"目录下创建"/dfs/data"目录
$mkdir -p tmp                       --在"/opt/Hadoop-2.6.0"目录下创建"/dfs/tmp"目录
```

所有的 Master 节点和 Slave 节点都进行上述操作后，Hadoop 系统就安装成功了，下一步将要配置 Hadoop 集群。

2. 配置 Hadoop

Hadoop 的配置文件根据版本的不同，配置的文件也不同。Hadoop 1.0 生态系统的

配置文件是放在 $HADOOP_HOME/conf 目录下，关键的配置文件在 src 目录都有对应的存放着默认值的文件，如表 3.9 所示。

表 3.9　Hadoop 1.0 生态系统配置文件

配 置 文 件	默认值配置文件
$HADOOP_HOME/conf/core-site.xml	$HADOOP_HOME/src/core/core-default.xml
$HADOOP_HOME/conf/hdfs-site.xml	$HADOOP_HOME/src/hdfs/hdfs-default.xml
$HADOOP_HOME/conf/mapred-site.xml	$HADOOP_HOME/src/mapred/mapred-default.xml

从表 3.9 可以看出，Hadoop 1.0 生态系统主要是对 core-site.xml、hdfs-site.xml 和 mapred-site.xml 文件的配置。到 Hadoop 2.0 生态系统时，Hadoop 架构发生了变化，而配置文件的路径和配置内容也相应发生了变化，其中配置文件目录变为 $HADOOP_HOME/etc/hadoop 目录，主要配置文件包括 hadoop-env.sh、yarn-env.sh、core-site.xml、hdfs-site.xml、mapred-site.xml 和 yarn-site.xml。本实例使用 Hadoop 2.6 版本，以该版本为例介绍需要配置的文件及配置内容。具体的配置过程如下。

1）配置所有 Slave 节点

该配置文件在/opt/hadoop-2.6.0/etc/hadoop/目录下的 slaves 文件，每行只添加一个 Slave 主机名，具体命令如下。

```
#vim /opt/hadoop-2.6.0/etc/hadoop/slaves
 Slave1.Hadoop
 Slave2.Hadoop
 Slave3.Hadoop
```

2）配置 hadoop-env.sh 和 yarn-env.sh

hadoop-env.sh 和 yarn-env.sh 主要用于设置 RescoureManager 等所需要的环境变量，主要需要修改 JAVA_HOME，即将 JDK 的安装目录添加到这两个文件中。这两个配置文件均在/opt/hadoop-2.6.0/etc/hadoop/目录下。本实例 JDK 的安装目录为/usr/java/jdk1.7.0_71，然后将环境变量加入 hadoop-en.sh 和 yarn-env.sh 文件中，具体命令如下。

```
$env|grep JAVA_HOME                                  --检查是否配置 JAVA_HOME
$su - root                                           --切换 root 用户
#vim /opt/hadoop-2.6.0/etc/hadoop/hadoop-env.sh      --添加 JAVA_HOME
export JAVA_HOME=/usr/java/jdk1.7.0_71
#vim /opt/hadoop-2.6.0/etc/hadoop/yarn-env.sh        --添加 JAVA_HOME
export JAVA_HOME=/usr/java/jdk1.7.0_71
```

3）配置 core-site.xml 文件

该文件是全局配置，主要配置的是 HDFS 的地址、端口号(有关 Hadoop 默认端口号请查看附表 1)和 hadoop.tmp.dir 参数。如果没有配置 hadoop.tmp.dir 参数，系统将默

认一个临时目录。core-site. xml 的主要配置参数如表 3. 10 所示。

表 3. 10 core-site. xml 主要配置参数

参 数	默 认 值	描 述
fs. defaultFS	0. 0. 0. 0:9000	hdfs://localhost:port/
io. file. buffer. size	131072	读写缓冲区大小
hadoop. tmp. dir	/tmp/hadoop- $ {user. name}	临时目录

core-site. xml 文件具体配置内容如下。

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
      <value>hdfs://master1.hadoop:9000</value>
  </property>
  <property>
    <name>io.file.buffer.size</name>
      <value>131702</value>
  </property>
  <property>
    <name>hadoop.tmp.dir</name>
      <value>file:/opt/hadoop-2.6.0/tmp</value>
  </property>
  <property>
    <name>hadoop.proxyuser.hadoop.hosts</name>
      <value></value>
  </property>
  <property>
    <name>hadoop.proxyuser.hadoop.groups</name>
      <value></value>
  </property>
</configuration>
```

4）配置 hdfs-site. xml 文件

该文件主要配置数据副本保存份数，以及 namenode、secondarynamenode 和 datanode 数据保存路径以及 http-address 等。hdfs-site. xml 的主要配置参数如表 3. 11 所示。

表 3. 11 hdfs-site. xml 主要配置参数

参 数	默 认 值	描 述
dfs. namenode. name. dir	$ {hadoop. tmp. dir}/dfs/name	表示 NameNode 存储命名空间和操作日志相关的元数据信息的本地文件系统目录

续表

参　　数	默　认　值	描　　述
dfs. datanode. data. dir	${hadoop. tmp. dir}/dfs/data	表示DataNode节点存储HDFS文件的本地文件系统目录
dfs. replication	3	指定block的副本数
dfs. namenode. secondary. http-address	0. 0. 0. 0:9001	表示SecondNameNode主机及端口号
dfs. webhdfs. enabled	true	开启WebHDFS

hdfs-site. xml文件具体配置内容如下。

```
<configuration>
    <property>
        <name>dfs.namenode.name.dir</name>
            <value>file:/opt/hadoop-2.6.0/dfs/name</value>
    </property>
    <property>
        <name>dfs.datanode.data.dir</name>
            <value>file:/opt/hadoop-2.6.0/dfs/data</value>
    </property>
    <property>
        <name>dfs.replication</name>
            <value>3</value>
    </property>
    <property>
        <name>dfs.namenode.secondary.http-address</name>
        <value>master1.hadoop:9001</value>
    </property>
    <property>
        <name>dfs.webhdfs.enabled</name>
            <value>true</value>
    </property>
</configuration>
```

5）配置mapred-site. xml文件

该文件主要配置使用Yarn计算框架和Jobhistory地址等。mapred-site. xml的主要配置参数如表3. 12所示。

表3. 12　mapred-site. xml主要配置参数

参　　数	默认值	描　　述
mapreduce. framework. name	local	执行MapReduce任务所使用的运行框架

续表

参　　数	默 认 值	描　　述
mapreduce. jobhistory. address	0.0.0.0:10020	配置 jobhistory 地址
mapreduce. jobhistory. webapp. address	0.0.0.0:19888	配置 Jobhistory Server Web UI 地址

mapred-site. xml 文件具体的配置内容如下。

```
#cp mapred-site.xml.template mapred-site.xml
<configuration>
    <property>
        <name>mapreduce.framework.name</name>
            <value>yarn</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.address</name>
            <value>master1.hadoop:10020</value>
    </property>
    <property>
        <name>mapreduce.jobhistory.webapp.address</name>
            <value>master1.hadoop:19888</value>
    </property>
</configuration>
```

6）配置 yarn-site. xml 文件

该文件主要配置 ResourceManager 地址及 yarn. application. classpath 等。yarn-site. xml 的主要配置参数如表 3.13 所示。

表 3.13　yarn-site. xml 主要配置参数

参　　数	默 认 值	描　　述
yarn. nodemanager. aux-services	mapreduce_shuffle	表示 MR Applicatons 所使用的 shuffle 工具类
yarn. nodemanager. auxservices. mapreduce. shuffle. class	org. apache. hadoop. mapred. ShuffleHandler	配置 mapreduce. shuffle
yarn. resourcemanager. address	0.0.0.0:8032	客户端对 Resource Manager 主机通过 host:port 提交作业
yarn. resourcemanager. scheduler. address	0.0.0.0:8030	ApplicationMasters 通过 Resource Manager 主机访问 host:port 跟踪调度程序获取资源
yarn. resourcemanager. resource-tracker. address	0.0.0.0:8031	Node Managers 通过 Resource Manager 主机访问 host:port
yarn. resourcemanager. admin. address	0.0.0.0:8033	管理命令通过 Resource Manager 主机访问 host:port

续表

参　　数	默　认　值	描　　述
yarn.resourcemanager.webapp.address	0.0.0.0:8088	Resource Manager Web 页面 host:port.
yarn.nodemanager.resource.memory-mb	8192	Node Manager 可用的物理内存

yarn-site.xml 文件具体配置内容如下。

```
<configuration>
    <property>
        <name>yarn.nodemanager.aux-services</name>
            <value>mapreduce_shuffle</value>
    </property>
    <property>
        <name>yarn.nodemanager.auxservices.mapreduce.shuffle.class</name>
            <value>org.apache.hadoop.mapred.ShuffleHandler</value>
    </property>
    <property>
        <name>yarn.resourcemanager.address</name>
            <value>master1.hadoop:8032</value>
    </property>
    <property>
        <name>yarn.resourcemanager.scheduler.address</name>
            <value>master1.hadoop:8030</value>
    </property>
    <property>
        <name>yarn.resourcemanager.resource-tracker.address</name>
            <value>master1.hadoop:8031</value>
    </property>
    <property>
        <name>yarn.resourcemanager.admin.address</name>
            <value>master1.hadoop:8033</value>
    </property>
    <property>
        <name>yarn.resourcemanager.webapp.address</name>
            <value>master1.hadoop:8088</value>
    </property>
    <property>
        <name>yarn.nodemanager.resource.memory-mb</name>
            <value>1024</value>
    </property>
</configuration>
```

在 Master1. Hadoop 节点安装并配置好 Hadoop 之后，可通过“scp -r /opt/hadoop root@服务器 IP:/opt/”将 Master1. Hadoop 上配置好的 hadoop 所在文件夹复制到所有的 Slave 的/opt/目录下。然后在各自机器上将 hadoop 文件夹权限赋予各自的 hadoop 用户，并且配置好环境变量等，这里不再赘述。

7）启动 Hadoop

启动 Hadoop 之前，先要格式化 HDFS 文件系统，在 Master1. Hadoop 节点上使用 hadoop 用户进行操作，具体的命令如下。

```
$ hadoop namenode - format                              --格式化 HDFS 文件系统
```

当出现如图 3.31 所示的提示信息时，表示 HDFS 文件系统格式化成功（只需格式化一次，下次启动就不用再格式化）。

```
15/02/02 20:05:00 INFO util.ExitUtil: Exiting with status 0
15/02/02 20:05:00 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master1.hadoop/192.168.85.100
************************************************************/
```

格式化成功

图 3.31　HDFS 文件系统格式化成功

HDFS 文件系统格式化成功后，就可以启动 Hadoop 了（在启动前要确认集群中所有机器的防火墙处于关闭状态），具体的命令如下。

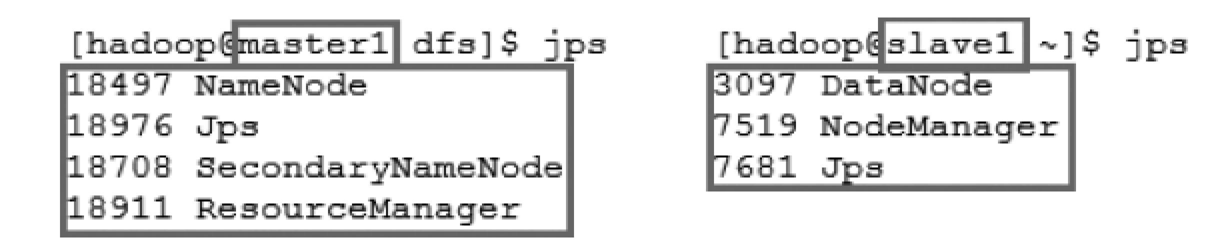
```
$ start- all.sh                                         --启动 Hadoop
```

start-all. sh 命令等同于 start-dfs. sh 和 start-yarn. sh 命令，将启动 DataNode、NameNode、SecondaryNameNode 和 NodeManager 角色，执行结果如图 3.32 所示。

```
[hadoop@master1 ~]$ start-all.sh
This script is Deprecated. Instead use start-dfs.sh and start-yarn.sh
Starting namenodes on [master1.hadoop]
master1.hadoop: starting namenode, logging to /opt/hadoop-2.6.0/logs/hadoop-hadoop-namenode-master1.hadoop.out
Slave1.Hadoop: starting datanode, logging to /opt/hadoop-2.6.0/logs/hadoop-hadoop-datanode-slave1.hadoop.out
Slave3.Hadoop: starting datanode, logging to /opt/hadoop-2.6.0/logs/hadoop-hadoop-datanode-slave3.hadoop.out
Slave2.Hadoop: starting datanode, logging to /opt/hadoop-2.6.0/logs/hadoop-hadoop-datanode-Slave2.Hadoop.out
Starting secondary namenodes [master1.hadoop]
master1.hadoop: starting secondarynamenode, logging to /opt/hadoop-2.6.0/logs/hadoop-hadoop-secondarynamenode-master1.hadoop.out
starting yarn daemons
starting resourcemanager, logging to /opt/hadoop-2.6.0/logs/yarn-hadoop-resourcemanager-master1.hadoop.out
Slave1.Hadoop: starting nodemanager, logging to /opt/hadoop-2.6.0/logs/yarn-hadoop-nodemanager-slave1.hadoop.out
Slave3.Hadoop: starting nodemanager, logging to /opt/hadoop-2.6.0/logs/yarn-hadoop-nodemanager-slave3.hadoop.out
Slave2.Hadoop: starting nodemanager, logging to /opt/hadoop-2.6.0/logs/yarn-hadoop-nodemanager-Slave2.Hadoop.out
```

图 3.32　成功启动 Hadoop

Hadoop 是否启动成功可以在终端输入“jps”命令查看进程进行验证，如图 3.33 所示查看 Master1. Hadoop 和 Slave1. Hadoop 运行的进程。



(a) Master1.Hadoop运行的进程　　(b) Slave1.Hadoop运行的进程

图 3.33　Master 节点和 Slave 节点运行进程

从图 3.33 可以看出，Master1. Hadoop 为 NameNode 节点，运行的进程有 NameNode、SecondaryNameNode 和 ResourceManager 进程；Slave1. Hadoop 为 Slave 节点，运行的进程有 DataNode 和 NodeManager 进程（集群规划参考表 3.7）。

注：Hadoop 2.6.0 具有 NodeManager Restart 功能，这个功能可以使 NodeManager 在不丢失运行在节点中的活动的 container 的情况下重新启动。如果在 Slave 节点没有看到 NodeManager 节点，很有可能是 NodeManager Restart 功能在起作用。

Hadoop 的验证也可以在浏览器地址栏中输入"http://192.168.85.100:8088/"查看集群信息，如图 3.34 所示。

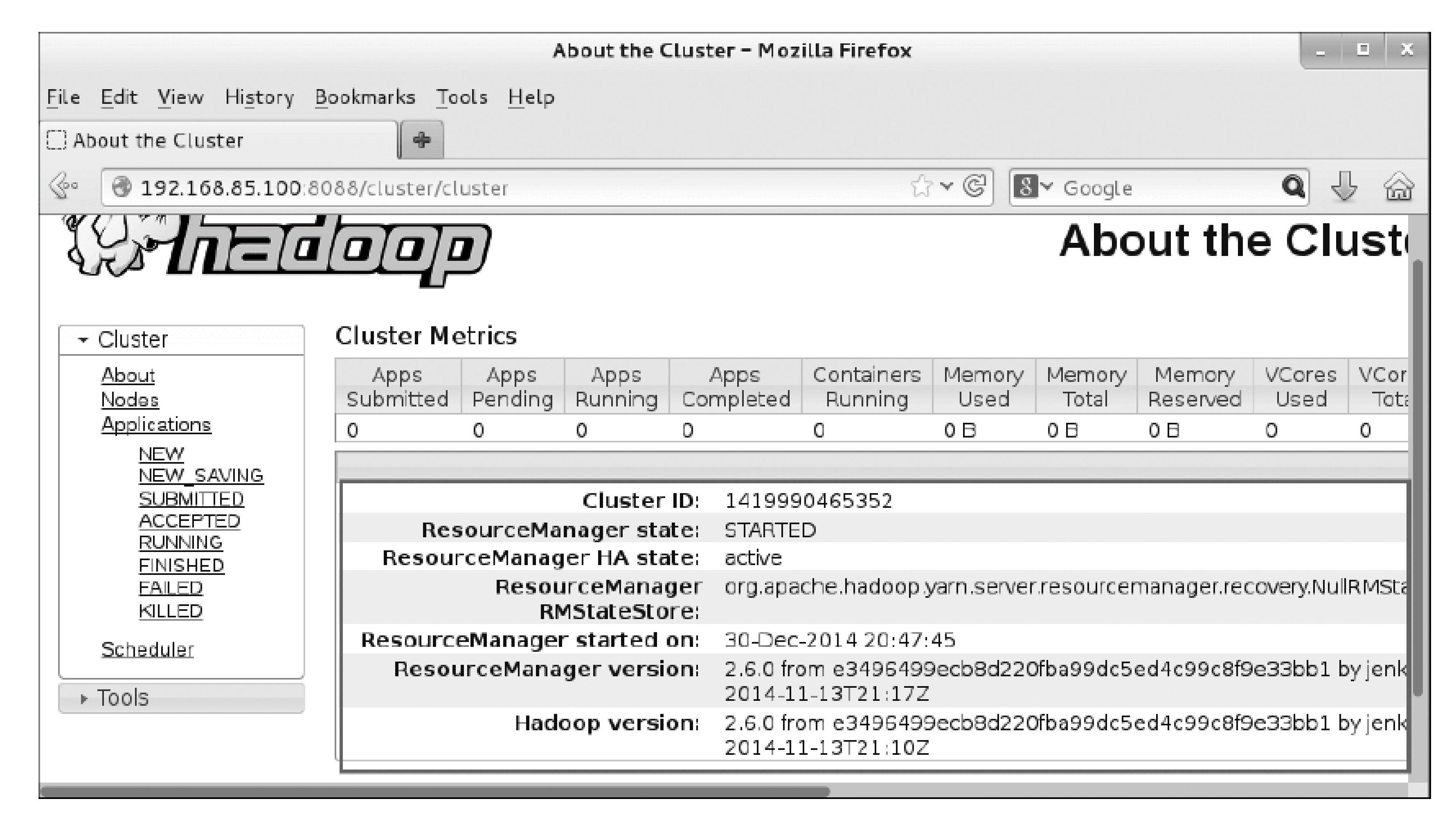


图 3.34　Hadoop 集群信息

注：有关 core-site.xml、hdfs-site.xml、mapred-site.xml 和 yarn-site.xml 配置文件的默认参考配置请查看本书配套资料中 Demo/Configuration_Default 文件夹下的 core-default.xml、hdfs-default.xml、mapred-default.xml 和 yarn-default.xml 配置文件。



第4章 分布式文件系统 HDFS

Hadoop 分布式文件系统 HDFS 是基于 Hadoop 大数据生态系统的底层核心，通过 HDFS 来实现对分布式存储的底层支持。本章将介绍分布式文件系统 HDFS 的主要特性、体系架构、存储原理等内容。

4.1 HDFS 概述

分布式文件系统(Distributed File System，DFS)是指文件系统管理的物理资源不一定存储在本地节点上，而是通过计算机网络与本地节点相连。Hadoop 作为一个分布式开源框架，引入了虚拟文件系统机制，并提供了一个文件系统抽象类(org. apache. hadoop. fs. FileSystem)。这个抽象类提供了对多种分布式文件系统的支持，如对 Hadoop 本身的分布式文件系统 HDFS 的支持以及对 Amazon S3 分布式文件系统的支持。对于分布式文件系统的选择要根据具体的业务场景来决定。HDFS 的设计目标是为了存储超大数据文件，而且提供了对数据的流式访问接口，适用于大规模数据处理。本节将对 Hadoop 分布式文件系统 HDFS 进行重点介绍。

Hadoop 分布式文件系统(Hadoop Distributed File System，HDFS)是 Hadoop 项目的核心子项目，是基于流数据模式访问和处理超大文件的需求而设计开发的，运行在通用硬件上，具有高容错性和高吞吐量，非常适合大规模数据集的分布式文件系统。它的开发和实现遵循了 Google 文件系统(GFS)的核心原理，并得到业界的极大关注和广泛应用，现在已经成为海量数据存储的事实标准。它的主要特点如下。

1. 硬件故障为常态现象

HDFS 部署在由普通商用机器组成的集群中，集群可能包含成百上千个机器节点，每个服务器节点上都存储着文件系统的部分数据，而节点故障是不可避免的。因此，HDFS 认为硬件故障为常态现象，并具有错误检测和快速、自动的恢复功能，即使某些节点发生故障时，整个集群的工作也不会受到影响。

2. HDFS 支持超大规模的数据集

运行在 HDFS 上的应用具有很大的数据集，典型文件大小从几十 GB 到几 TB，整个 HDFS 文件系统的数据量可以达到几十 PB。因此，HDFS 被设计为支持大文件存储，而

且通过 Yahoo 的实验验证，HDFS 可以支持几千个节点组成的集群，支持千万个数据文件。

3. 简单的一致性模型

HDFS 简化了传统的文件访问模型，它假定当一个文件被创建、写入并关闭后就不会再修改。这种文件访问模型为“简单的一致性”模型，即一次写入多次读取的文件访问模型。这种模型很容易处理数据的一致性问题，并大大提高了数据访问的吞吐量。

4. 流式数据访问

HDFS 提供了类似于流式的数据访问模式，HDFS 中节点之间的数据传输和访问的重点并不是数据访问的反应时间，而是数据吞吐量。因此，HDFS 的设计更多考虑到了数据的批量处理，而不是用户的交互处理，节点将数据以较小的数据包形式进行传输，从而提高了数据访问的吞吐量，有效避免了由于大量数据同时出现在信道上所造成的网络阻塞，也便于本地文件系统处理数据。

5. 高度容错性

HDFS 提供了很强的容错处理能力，将文件系统故障（服务器、网络、存储故障等）视为常态。HDFS 将大文件分割成很多文件块分开存储，并采用了完全备份的策略，每个文件块的备份数量默认为三个，一旦发生数据校验错误将重新进行复制。这种备份策略，不仅保证了数据的完整性，而且也使用户能够就近访问数据，缩短了数据的传输时间。

6. 高可扩展性

HDFS 将文件的数据块分配信息存储在 NameNode 节点上，文件数据块信息分布在 DataNode 节点上，当整个系统容量需要扩充时，只需要动态地增加 DataNode 节点数据，HDFS 将会自动地实时将新的 DataNode 节点信息加入到 NameNode 之中，整个过程不会对系统的稳定性和用户的使用造成影响。

7. 平台移植性

HDFS 在设计时就考虑到了平台的可移植性。因此，HDFS 采用 Java 语言开发，具有良好的平台移植性，它不仅可以运行在 Linux、Windows 等操作系统上，而且可以方便地从一个平台移植到另一个平台。这种特性方便了 HDFS 作为大规模数据应用平台的推广。

HDFS 也并不是通用的分布式文件系统，在处理一些特定问题时，不但没有优势，而且有一定的局限性，主要局限性体现在以下几个方面。

1）低延迟数据访问

HDFS 设计之初的目的主要是为了达到较高的数据吞吐量，这是以高延迟为代价的。因此，HDFS 不适用于处理对数据访问要求低延迟的场景。对于这种情况，可以通过上层的数据管理工具来尽可能地弥补不足（如使用 HBase），或者使用缓存、多 Master 设计

也可以降低客户端的数据请求压力,从而减少延时。

2) 存储大量小文件

HDFS中每个文件都要在NameNode中记录其实际存储位置,即元数据(文件的基本信息),并且为了提高读取速度,元数据存储在NameNode的内存中,所以文件系统所能容纳的文件数目是由NameNode的内存大小来决定。一般来说,每一个文件、文件夹和Block需要占据150B左右的空间(100万个文件就需要300MB空间),如果小文件太多可能使元数据信息过大,导致NameNode内存无法容纳所有元数据。因此,HDFS并不适合存储大量小文件。对于这种情况,可以利用SequenceFile、MapFile、Har等方式归档小文件,或者采用多Master设计的方式来解决。

3) 多用户写入或修改文件

在HDFS中的文件中只有一个写入者,而且写操作只能将数据添加在文件的末尾,即只能执行追加操作。因此,HDFS还不支持多个用户对同一文件的写操作,以及在文件任意位置进行修改。

因此,HDFS适用于存储数据量巨大,存储的单个文件也大(GB/TB级别),数据处理以流式读为主,对数据访问并不要求低延迟,追求数据访问的高吞吐量,而且对硬件故障有一定容忍度的场景中。

4.2 HDFS基本组成

HDFS主要有数据块(Block)、数据节点(DataNode)、元数据节点(NameNode)和辅助元数据节点(SecondaryNameNode)等几个重要的组成部分。只有正确理解每一部分在HDFS中的角色,才能更深刻地理解HDFS。

421 数据块

传统文件系统是基于存储块进行操作的,HDFS也使用了块的概念,因此,HDFS默认的最基本的存储单位是数据块(一般为64MB或128MB的数据块)。HDFS使用块作为存储和操作单元所带来的好处如下。

(1) 大大简化存储子系统的设计;

(2) 更有利于分布式文件系统中数据容错能力和可用性的实现;

(3) 对于存储任意大的文件不受任一单个节点磁盘大小的限制;

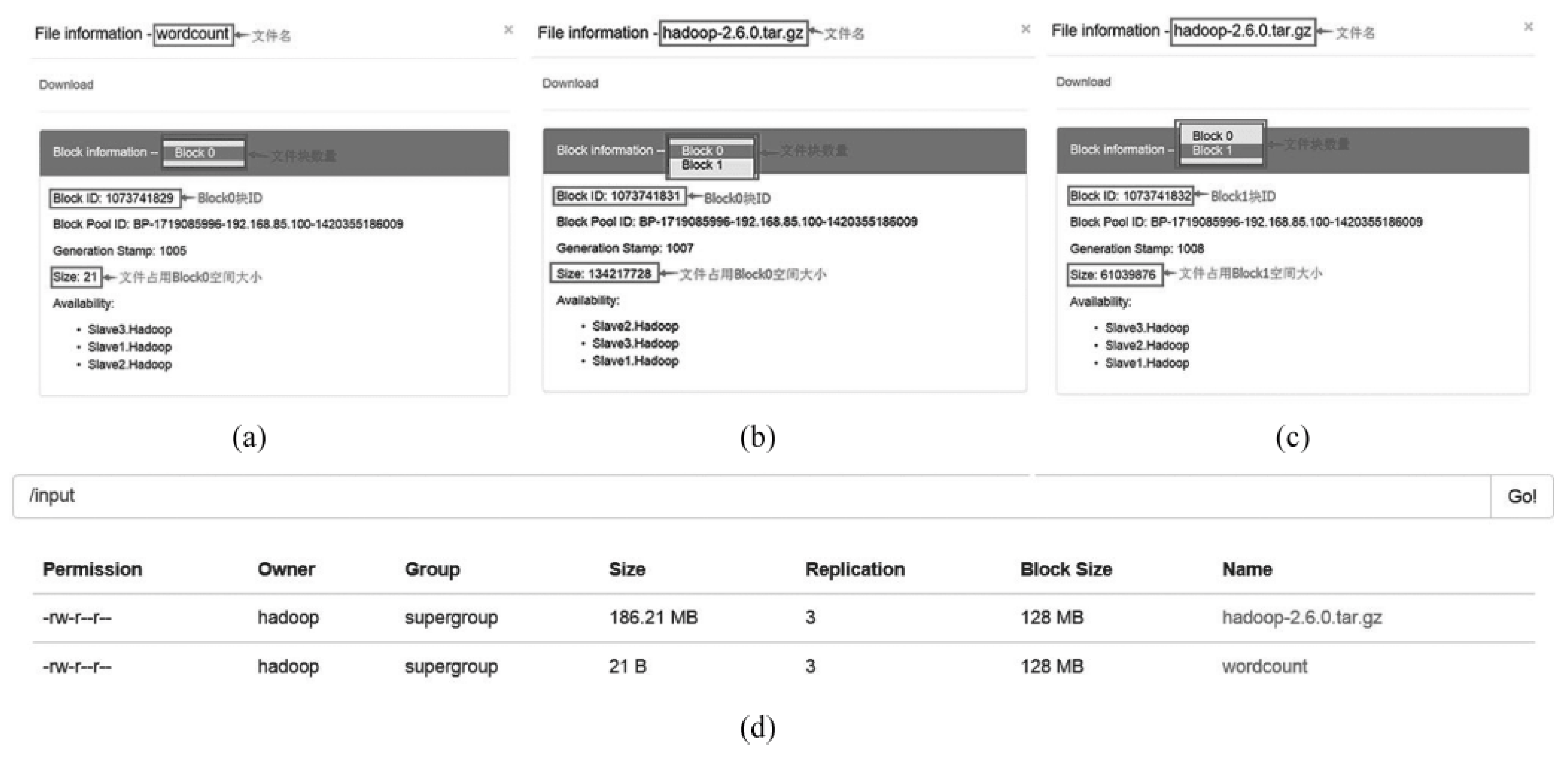
(4) 块集中存储减少磁盘寻道次数,进而减少了寻道时间;

(5) 数据以块方式存储有更高的安全性。

HDFS将一个文件以块为单位在集群服务器上分配存储,而且每个块都有一个自己的全局ID。一个文件有可能包含多个块,一个块也可以包含多个文件,由文件的大小和块大小的参数决定(在hdfs-site.xml中的dfs.blocksize参数设置块大小)。一般情况下,如果一个文件大于数据块的大小,HDFS将大文件分解得到若干个数据块;如果一个文件小于数据块的大小,则按该文件的实际大小组块存储,如图4.1所示。

从图4.1中可以看出,文件wordcount和hadoop-2.6.0.tar.gz在HDFS上所存储

的块信息。其中，图 4.1(d)显示了两个文件的大小分别为 21B 和 186.21MB，块的大小设置为 128MB，并且这两个文件都有三个副本；图 4.1(a)显示了文件 wordcount 在 HDFS 上的块数量(一个块 Block0)，文件占用 Block0 空间大小及块 ID 等信息；图 4.1(b)和图 4.1(c)显示了文件 hadoop-2.6.0.tar.gz 的块数量(两个块，分别是 Block0 和 Block1)，文件占用块空间的大小及块 ID 等信息。由于文件 wordcount 小于数据块大小(128MB)，因此该文件在 HDFS 上占用一个数据块；文件 hadoop-2.6.0.tar.gz 大小为 186.21MB 大于数据块大小(128MB)，因此该文件在 HDFS 上占用了两个数据块。



Permission	Owner	Group	Size	Replication	Block Size	Name
-rw-r--r--	hadoop	supergroup	186.21 MB	3	128 MB	hadoop-2.6.0.tar.gz
-rw-r--r--	hadoop	supergroup	21 B	3	128 MB	wordcount

(d)

图 4.1　HDFS 中文件块信息

422　元数据节点

所谓元数据(Metadata)是指数据的数据，HDFS 与传统的文件系统一样，提供了一个分级的文件组织形式，维护这个文件系统所需的信息(除了文件本身内容)就称为 HDFS 的元数据。元数据节点(NameNode)是用来管理文件系统的命名空间(Namespace)。该命名空间中含有定位文件的某个数据块所需的所有信息，如 DataNode 列表、数据块列表、文件和目录列表等元数据信息，其中文件和目录以 iNode 的方式描述。

元数据节点(NameNode)将与目录和文件(即命名空间 Namespce)有关的信息存储在内存中，并且主要维持两张表：<filename，blocksequence>和<block，machinelist>。通过第一张表可以找到文件所用的数据块 ID(block id)，通过第二张表可以找到数据块的实际物理存储位置。这些信息同样也会保存到本地硬盘的以下文件中(Hadoop 1.0 生态系统和 Hadoop 2.0 生态系统保存在本地的文件名相同)。

1. fsimage

fsimage 文件(是一个二进制文件)及其对应的 md5 校验文件，保存了文件系统目录树信息，以及文件和块的对应关系信息，是 HDFS 中元数据相关的重要文件。在 Hadoop

1.0 生态系统中，fsimage 信息不是最新的；而在 Hadoop 2.0 生态系统中，fsimage 信息是最新的。图 4.2 给出了本实例 HDFS 上的 fsimage 信息（本实例的 fsimage 及对应的 md5 校验文件在/opt/hadoop-2.6.0/dfs/name/current/目录下）。

2. edits

edits 文件中保存了最新检查点后的命名空间的变化。当客户端对文件进行读写操作时，操作首先会记入到 edits 文件中，然后才会更改内存中的数据。在 Hadoop 1.0 生态系统中，edits 信息不是最新的；而在 Hadoop 2.0 生态系统中，edits 信息是最新的。图 4.3 给出了本实例 HDFS 上的部分 edits 信息。

```
fsimage_0000000000000000222
fsimage_0000000000000000222.md5
fsimage_0000000000000001097
fsimage_0000000000000001097.md5
```

图 4.2　HDFS 上的 fsimage 及对应的 md5 检验文件

```
edits_0000000000000000476-0000000000000000477
edits_0000000000000000478-0000000000000000479
edits_0000000000000000480-0000000000000000481
edits_0000000000000000482-0000000000000000483
edits_0000000000000000484-0000000000000000485
```

图 4.3　HDFS 上的 edits 文件

3. VERSION

VERSION 文件是 Java 的属性文件，保存了 HDFS 的版本号等信息。下面以本实例 HDFS 上的 VERSION 文件为例（本实例的 VERSION 文件在/opt/hadoop-2.6.0/dfs/name/current/目录下），其内容如下。

```
#Tue Jan 06 21:15:19 EST 2015
namespaceID=143058597
clusterID=CID-447ae62c-1420-4811-902c-c802b20d1338
cTime=0
storageType=NAME_NODE
blockpoolID=BP-1719085996-192.168.85.100-1420355186009
layoutVersion=-60
```

在 VERSION 文件中，namespaceID 是文件系统的唯一标识符，是在文件系统初次格式化时生成的；clusterID 是系统生成或手动指定的集群 ID，在-clusterid 选项中可以使用它；cTime 表示 NameNode 存储时间的创建时间，如果 NameNode 没有更新过，记录值将设为 0，否则 cTime 将会记录更新时间戳；storageType 说明这个文件存储的是什么进程的数据结构信息，如果是 NameNode，storageType＝NAME_NODE；blockpoolID 是针对每一个 Namespace 所对应的 blockpool 的 ID，ID 内容包括与其对应的 NameNode 节点的 IP 地址等信息；layoutVersion 表示 HDFS 永久性数据结构的版本信息，只要数据结构变更，版本号也要递减，此时的 HDFS 也需要升级，否则磁盘仍旧是使用旧版本的数据结构，这会导致新版本的 NameNode 无法使用。

4. seen_txid

seen_txid 是用来存放 transactionId 的文件。如果将 HDFS 格式化之后，seen_txid



文件中的内容是 0，它代表的是 NameNode 里面的 edits_ * 文件的尾数。当 NameNode 重启时，HDFS 会按照 seen_txid 所提供的数字，循序从 edits_0000001 到 seen_txid 的数字。下面以本实例 HDFS 上的 seen_txid 文件为例（seen_txid 文件在/opt/hadoop-2.6.0/dfs/name/current/目录下），其内容如下：

```
1583
```

当 NameNode 重启时，HDFS 会按照 seen_txid 所提供的数字 1583，循序从 edits_0000001 到 1583，如图 4.4 所示。

```
edits_0000000000000001567-0000000000000001568
edits_0000000000000001569-0000000000000001570
edits_0000000000000001571-0000000000000001572
edits_0000000000000001573-0000000000000001574
edits_0000000000000001575-0000000000000001576
edits_0000000000000001577-0000000000000001578
edits_0000000000000001579-0000000000000001580
edits_0000000000000001581-0000000000000001582
edits_inprogress_0000000000000001583
fsimage_0000000000000000222
fsimage_0000000000000000222.md5
fsimage_0000000000000001097
fsimage_0000000000000001097.md5
seen_txid
VERSION
```

edits尾

seen_txid文件中的内容为：1583

图 4.4　seen_txid 文件

如果 seen_txid 内的数字不是 edits 最后的尾数，就会导致启动 NameNode 时元数据信息缺失，发生误删 DataNode 上数据块的情况。

上述这些文件的默认存储路径为：${dfs.namenode.name.dir}/current/目录下，其中 dfs.namenode.name.dir 是在 hdfs-site.xml 文件中进行配置的，默认值如下：

```
<property>
  <name>dfs.namenode.name.dir</name>
  <value>file://${hadoop.tmp.dir}/dfs/name</value>
  <description>Determines where on the local filesystem the DFS name node
      should store the name table(fsimage).  If this is a comma-delimited list
      of directories then the name table is replicated in all of the
      directories, for redundancy.</description>
</property>
```

在 Hadoop 2.0 生态系统中，根据 dfs.namenode.name.dir 的<description>属性描述，NameNode 目录可以配置多个，如/data1/dfs/name、/data2/dfs/name、/data3/dfs/name、…。各个目录存储的文件结构和内容都完全一样，即使其中一个目录损坏，也不会影响 HDFS 中的元数据。

当一个 NameNode 启动时，它首先从一个映像文件 fsimage 中读取 HDFS 的状态，并应用日志文件中的 edits 操作。然后，将最新的 HDFS 状态写入 fsimage 中，并使用一个空的 edits 文件（edits.new）开始正常操作。因为 NameNode 只有在启动阶段才合并 fsimage 和 edits，所以一段时间后日志文件可能会变得非常庞大，特别是对大型的集群。

日志文件太大的另一个副作用是下一次 NameNode 启动会花很长时间。

4.2.3 辅助元数据节点

在 Hadoop 1.0 生态系统中，辅助元数据节点(SecondaryNameNode)是对元数据节点(NameNode)的一个补充，本质上是 NameNode 的一个快照，其主要功能就是周期性地将元数据节点的命名空间镜像文件 fsimage 和修改日志 edits 合并，以防日志文件 edits 过大，导致元数据节点(NameNode)启动时间过长，并将合并过后的命名空间镜像文件也在辅助元数据节点保存了一份，从而防止元数据节点失败所导致的数据丢失问题。在 Hadoop 2.0 生态系统中，已经通过 HA 机制解决了 NameNode 的单点故障问题，同时不再用 SecondaryNameNode 对 fsimage 和 edits 合并了(Hadoop 2.0 生态系统如何实现 fsimage 和 edits 合并，请查看 HDFS HA 章节)。下面将重点介绍在 Hadoop 1.0 生态系统中的辅助元数据节点(SecondaryNameNode)。

辅助元数据节点(SecondaryNameNode)定期从元数据节点(NameNode)上下载元数据信息(元数据镜像 fsimage 和元数据库操作日志 edits)，然后将 fsimage 和 edits 进行合并，生成新的 fsimage.ckpt(该 fsimage 就是 SecondaryNameNode 下载时刻的元数据的 Checkpoint)保存在本地，同时将其推送到 NameNode，定期合并 fsimage 和 edits 日志，把 edits 日志文件大小控制在一个限度下，其合并机制如图 4.5 所示。

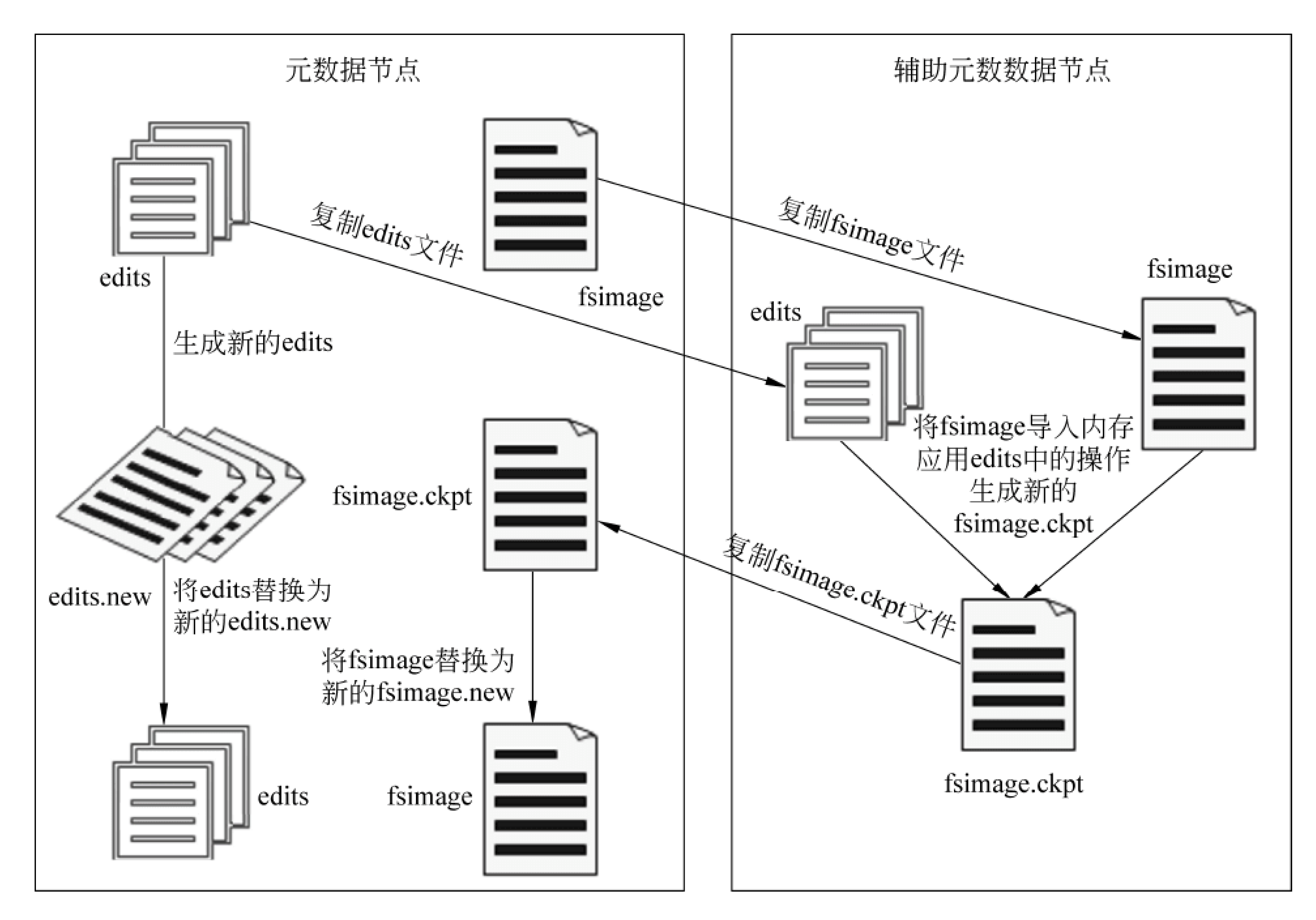


图 4.5 fsimage 和 edits 合并机制

从图 4.5 可以看出，辅助元数据节点(SecondaryNameNode)通知元数据节点(NameNode)生成新的日志文件 edits(目的合并 edits)，并从 NameNode 获得 fsimage 文

件和旧的 edits 文件。然后，SecondaryNameNode 将 fsimage 载入内存，并执行日志文件 edits 中的操作，从而生成新的 fsimage 文件（fsimage. ckpt）。最后，SecondaryNameNode 将新的 fsimage. ckpt 文件发回给 NameNode，NameNode 将旧的 fsimage 文件及旧的 edits 文件替换为新的 fsimage 文件和新的 edits 文件并写入此次检查时间。在 fsimage 和 edits 合并过程中，SecondaryNameNode 对内存的需求与 NameNode 是相同的，所以对于那些大型的生产系统中，如果将两者部署到同台服务器上，在内存上会出现瓶颈。所以最好将 NameNode 和 SecondaryNameNode 分别部署到不同的服务器。另外，SecondaryNameNode 保存的只是检查时刻的元数据，一旦 NameNode 上的元数据损坏，通过检查点恢复的元数据并不是 HDFS 此刻的最新数据，因此存在数据一致性的问题（在 Hadoop 2.0 生态系统中，fsimage 和 edits 都为最新数据，不存在数据一致性问题）。

注：在 Hadoop 1.0 生态系统中，检查周期的设定可在 core-site. xml 配置文件中对 fs. checkpoint. period 和 fs. checkpoint. size 进行设置。fs. checkpoint. period 表示指定连续两次检查点的最大时间间隔，默认值为 3600s；fs. checkpoint. size 定义了 edits 日志文件的最大值，默认值为 64MB。

4.2.4　数据节点

在分布式文件系统中，每个数据文件都被切分成若干个数据块，每个数据块都被存储在不同的服务器上。因此，存放数据块的服务器称为数据节点（DataNode）。数据节点是 HDFS 中真正存储数据块的地方，并且数据节点主要维持一张＜block，block_size＞表。DataNode 定期将该表上报给 NameNode，NameNode 通过该表来了解当前 DataNode 的空间使用情况。这些信息同样也会保存到本地硬盘的文件中。Hadoop 1.0 生态系统与 Hadoop 2.0 生态系统保存到本地的文件名会有所区别，但存储信息内容基本一致。

在 Hadoop 1.0 生态系统中，数据块信息保存到本地的以下文件中。

1）blk_＜id＞

blk_＜id＞保存的是 HDFS 的数据块，其中保存了具体的二进制数据。

2）blk_＜id＞. meta

blk_＜id＞. meta 保存的是数据块的属性信息，包括版本信息、类型信息等内容。

3）VERSION

VERSION 文件是 Java 的属性文件，保存了 HDFS 的版本号等信息。

在 Hadoop 2.0 生态系统中，数据块信息保存到本地的以下文件中。

1）BP-＜id＞

BP-＜id＞文件提供了一个 BlockPoolID 标识，并且是跨集群的全局唯一。当一个新的 Namespace 被创建时，会创建并持久化一个唯一 ID，并且 BlockPoolID 会被持久化到磁盘中，在后续的启动过程中，会再次被调入使用。全局唯一的 BlockPoolID 比人为的配置更可靠一些。

2）VERSION

VERSION 文件是 Java 的属性文件，保存了 HDFS 的版本号等信息。下面以本实例 HDFS 上的 VERSION 文件为例（本实例的 VERSION 文件是以 Slave3. Hadoop 节点中

的文件为例,在/opt/hadoop-2.6.0/dfs/data/current/目录下),其内容如下。

```
#Thu Jan 08 00:14:47 EST 2015
storageID=DS-d6a83e27-fd85-46ae-a4c4-c19f6db26611
clusterID=CID-447ae62c-1420-4811-902c-c802b20d1338
cTime=0
datanodeUuid=111ee1ee-c2e2-4c02-b9f0-c1f2548a0ff8
storageType=DATA_NODE
layoutVersion=-56
```

DataNode节点目录中的VERSION文件比NameNode节点目录中的VERSION文件多一个storageID属性和datanodeUuid属性。其中,storageID表示一个数据节点在集群中的唯一标识,是DataNode节点向NameNode节点第一次注册时,NameNode为它分配的一个分布式存储器标识,一个DataNode节点中所有存储路径的storageID是一样的;datanodeUuid表示DataNode的通用唯一识别码。

上述这些文件的默认存储路径为${dfs.datanode.data.dir}/current/目录下,其中dfs.datanode.data.dir是在hdfs-site.xml文件中进行配置的,默认值如下。

```
<property>
  <name>dfs.datanode.data.dir</name>
  <value>file://${hadoop.tmp.dir}/dfs/data</value>
  <description>Determines where on the local filesystem an DFS data node
  should store its blocks.  If this is a comma-delimited
  list of directories, then data will be stored in all named
  directories, typically on different devices.
  Directories that do not exist are ignored.
  </description>
</property>
```

根据dfs.datanode.data.dirr的<description>属性描述,DataNode目录同样也可以配置多个,如/data1/dfs/data、/data2/dfs/data、/data3/dfs/data、…。

另外,HDFS客户端(Client)在HDFS中也承担着重要的角色,是指需要访问HDFS文件服务的用户或应用,如命令行客户端、API客户端等。客户端的主要功能就是接收用户的请求,并通过网络与NameNode和DataNode交互等。

4.3 HDFS体系架构

Hadoop 1.0生态系统中HDFS和Hadoop 2.0生态系统中的HDFS体系架构有着明显的不同。下面将分别介绍在不同生态系统中HDFS的体系架构。

4.3.1 Hadoop 1.0生态系统中HDFS体系架构

在Hadoop 1.0生态系统中,HDFS的体系架构是一个典型的Master/Slave架构,包括一个NameNode节点和多个DataNode节点,并提供应用程序访问接口Client。

Hadoop 1.0 生态系统中 HDFS 体系架构如图 4.6 所示。

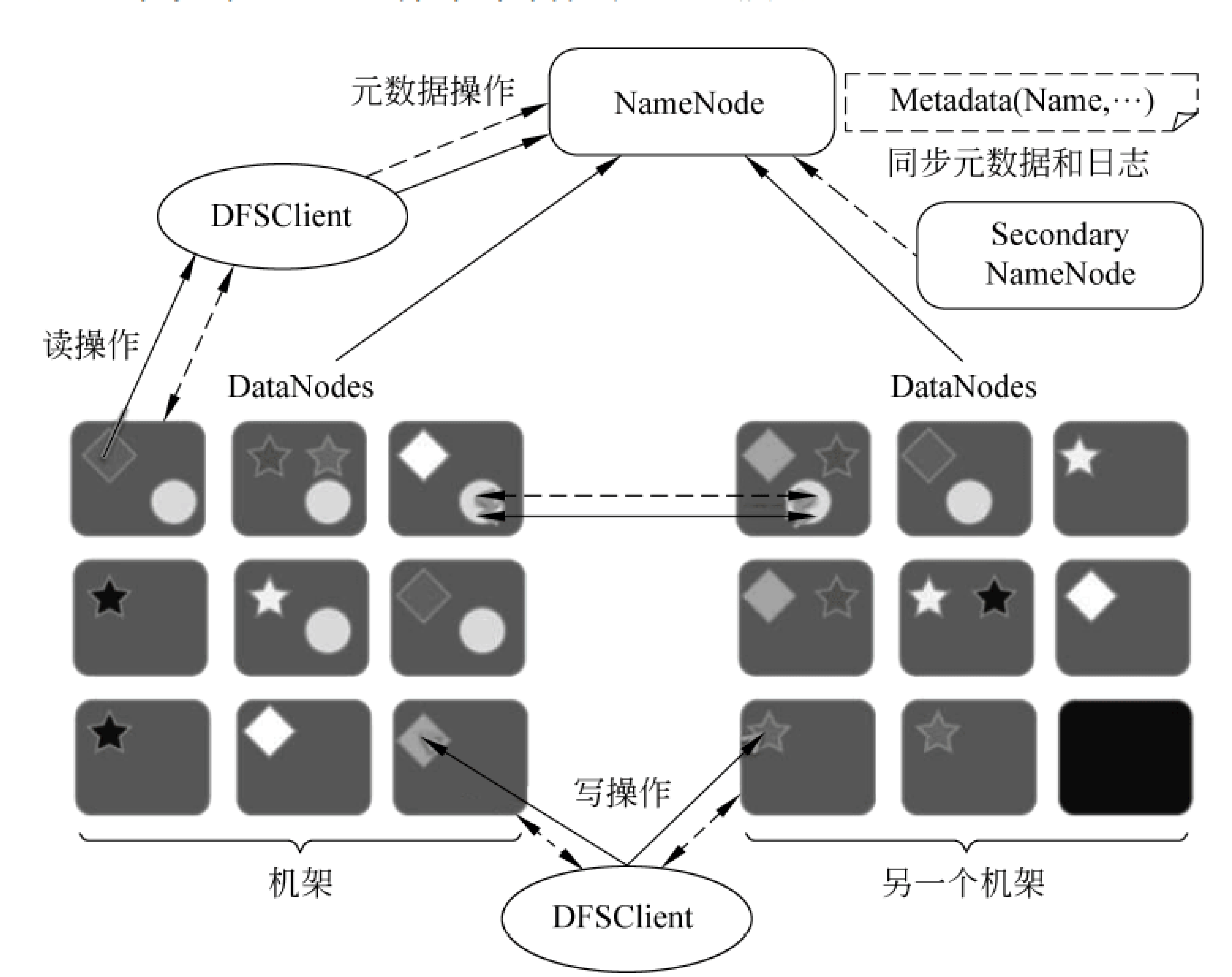


图 4.6　Hadoop 1.0 生态系统中 HDFS 的体系架构

图 4.6 中的连线表示两者之间存在通信，箭头一方表示请求，没有箭头的一端表示发起请求的一方；图中的实线表示数据消息的通路，虚线表示控制消息的通路。HDFS 的通信分为控制通信和数据通信两种，而且所有的 HDFS 通信协议都是构建在 TCP/IP 上。HDFS 上 NameNode、DataNode 和 Client 之间的通信主要有 4 种，即 Client 与 NameNode、Client 与 DataNode、NameNode 与 DataNode、DataNode 与 DataNode。其中，Client 通过一个可配置的端口连接到 NameNode，通过 Client Protocol 与 NameNode 交互；DataNode 是使用 DataNode Protocol 与 NameNode 交互；NameNode 不会主动发起通信请求，而是响应来自 Client 和 DataNode 的请求。

从图 4.6 可以看出，NameNode 是整个文件系统的管理节点，它负责文件系统名字空间(Namespace)的管理与维护，同时负责客户端文件操作的控制以及具体存储任务的管理与分配；DataNode 提供真实文件数据的存储服务；Client 负责与 NameNode 和 DataNode 交互。HDFS 将文件的数据块分配信息存放在 NameNode 服务器之上，文件数据块的信息分布地存放在 DataNode 服务器上。当整个系统容量需要扩充时，只需要增加 DataNode 的数量，系统会自动地实时将新的服务器匹配进整体阵列之中。然后，文件的分布算法会将数据块搬迁到新的 DataNode 之中，不需任何系统宕机维护或人工干预。然而 HDFS 无法高效地存储大量小文件，也不支持多用户写入及任意修改文件的操作。

4.3.2　Hadoop 2.0 生态系统中 HDFS 体系架构

与 Hadoop 1.0 生态系统中的 HDFS 相比，Hadoop 2.0 生态系统中的 HDFS 增加了两个重大特性：HA 和 Federaion。HA 即为 High Availability，用于解决 NameNode 单

点故障问题，该特性通过热备的方式为主 NameNode 提供一个备用者，一旦主 NameNode 出现故障，可以迅速切换至备用 NameNode，从而实现不间断对外提供服务。Federation 即为“联邦”，该特性允许一个 HDFS 集群中存在多个 NameNode 同时对外提供服务，这些 NameNode 分管一部分目录（水平切分），彼此之间相互隔离，但共享底层的 DataNode 存储资源。Hadoop 2.0 生态系统中的 HDFS 体系架构如图 4.7 所示。

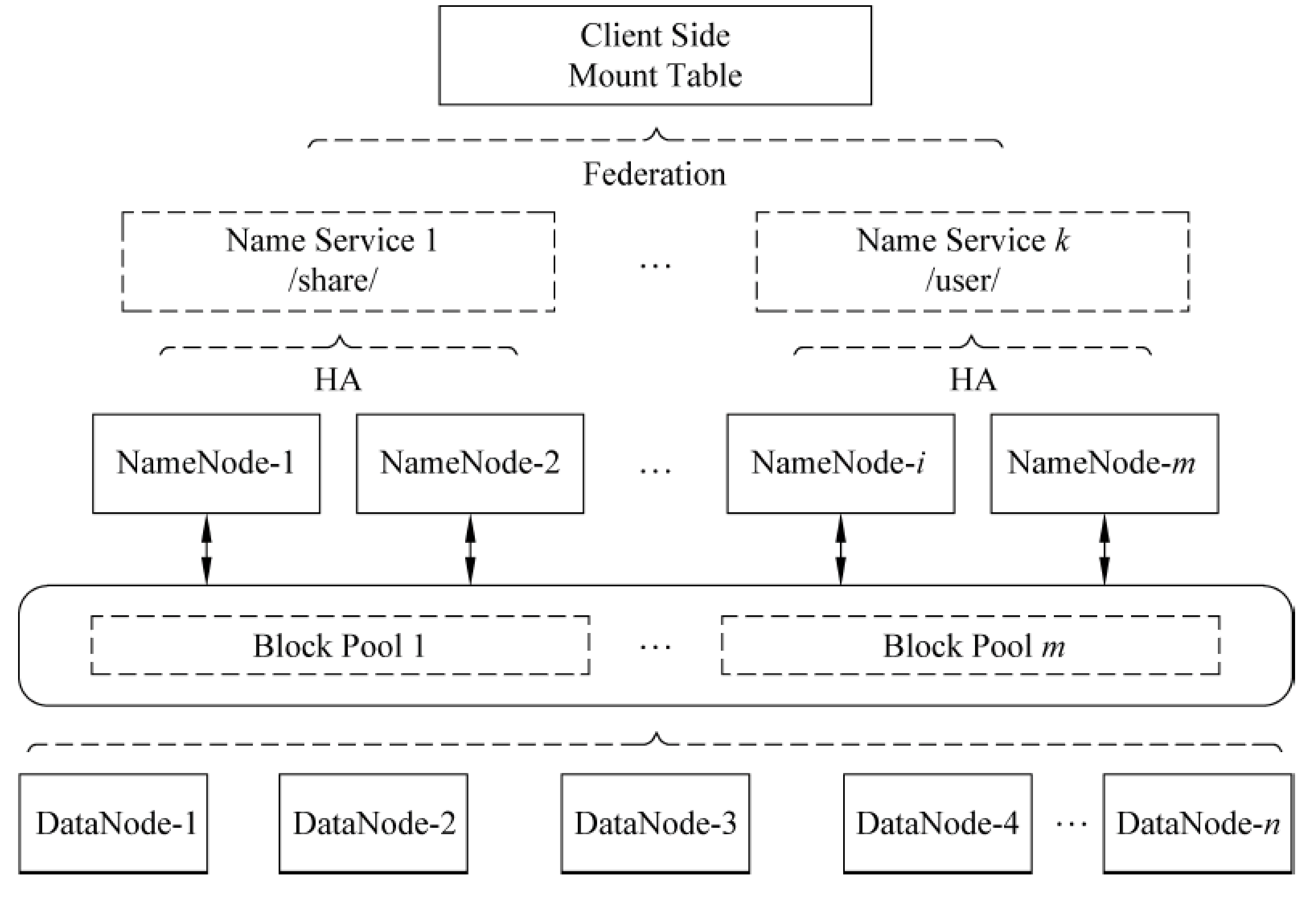


图 4.7 Hadoop 2.0 生态系统中 HDFS 的体系架构

从图 4.7 可以看出，Hadoop 2.0 生态系统中 HDFS 比 Hadoop 1.0 生态系统中的 HDFS 多出“Block Pool”和“Name Service”两个概念。Block Pool 是一个重新将 block 划分的逻辑概念，即属于一个命名空间的一组 block，每个 DataNode 会为多个 Block Pool 存储 block；Name Service 是 NameNode 的抽象，提供服务的不再是 NameNode 本身，而是 Name Service。在 HDFS 中可以有多个 Block Pool，每个 Block Pool 会各自管理各自的 block，它们之间相互独立且不需要互相协调；在 HDFS 中也可以有多个 Name Service，每个 Name Service 又是由一个或两个 NameNode 组成。Hadoop 客户端 Client 是使用 Client Side Mount Table 方式做到数据共享和访问的，其原理是将各个命名空间挂载到全局 mount-table 中，实现数据全局共享，再将同样的命名空间挂载到 Client 的 mount-table 中，实现应用程序可见的命名空间视图。总之，Hadoop 2.0 生态系统中的 HDFS 通过 Federation 的形式实现了 NameNode 的横向扩展，并支持多个命名空间，同时通过 HA 机制实现了 NameNode 单节点故障。

4.4 HDFS 核心功能

HDFS 是 Hadoop 生态系统中最主要的分布式文件系统，负责管理文件系统元数据 NameNode 和存储实际数据的 DataNode。Hadoop 生态系统中的其他组件的更改和新特

性都有规律地遵循 HDFS。HDFS 中的一些比较重要的功能如表 4.1 所示。

表 4.1　HDFS 核心功能

核心功能	说　明
命名空间 Namespace	HDFS 支持传统的层次型文件组织,与大多数其他文件系统类似,用户可以创建目录,并在其间创建、删除、移动和重命名文件
Shell 命令	Hadoop 包括一系列的类 Shello 命令,可直接和 HDFS 以及其他 Hadoop 支持的文件系统进行交互
数据复制	HDFS 被设计成在一个大集群中可以跨机器地可靠地存储海量的文件。它将每个文件存储成 block 序列,除了最后一个 block,所有的 block 都是同样的大小。文件的所有 block 为了容错都会被复制。每个文件的 block 大小和 replication 因子都是可配置的。replication 因子可以在文件创建的时候配置,以后也可以改变
机架感知	HDFS 的存放策略是将一个副本存放在本地机架上的节点,一个副本放在同一机架上的另一个节点,最后一个副本放在不同机架上的一个节点。机架的错误远远比节点的错误少,这个策略不会影响到数据的可靠性和有效性
集群均衡	HDFS 支持数据的均衡计划,如果某个 DataNode 节点上的空闲空间低于特定的临界点,那么就会启动一个计划自动地将数据从一个 DataNode 搬移到空闲的 DataNode。当对某个文件的请求突然增加,那么也可能启动一个计划创建该文件新的副本,并分布到集群中以满足应用的要求
数据完整性	从某个 DataNode 获取的数据块有可能是损坏的,这个损坏可能是由于 DataNode 的存储设备错误、网络错误或者软件 Bug 造成的。HDFS 客户端软件实现了 HDFS 文件内容的校验和。当某个客户端创建一个新的 HDFS 文件,会计算这个文件每个 block 的校验和,并作为一个单独的隐藏文件保存这些校验和在同一个 HDFS Namespace 下。当客户端检索文件内容,它会确认从 DataNode 获取的数据跟相应的校验和文件中的校验和是否匹配,如果不匹配,客户端可以选择从其他 DataNode 获取该 block 的副本
快照	快照支持某个时间的数据备份,当 HDFS 数据损坏的时候,可以恢复到过去一个已知正确的时间点。Hadoop 1.0 生态系统中 HDFS 不支持快照功能。Hadoop 2.0 生态系统中 HDFS 支持快照功能
空间的回收	删除文件并没有立刻从 HDFS 中删除,HDFS 将这个文件重命名,并转移到/trash 目录。当被删除的文件还保留在/trash 目录中的时候,如果用户想恢复这个文件,可以检索浏览/trash 目录并检索该文件。/trash 目录仅保存被删除文件的最近一次备份

4.5　HDFS 通信机制

在 Hadoop 集群中,为了方便集群中各组件之间的通信,Hadoop 采用了基于 IPC 模型实现的一个高效的轻量级 RPC 框架(Remote Procedure Call Protocol),提供了分布式环境下的对象调用功能。Hadoop 的 RPC 采用 Client/Server 模式,其中请求服务的一端为 Client,提供服务的一端为 Server,通信形式有 Client 和 NameNode(其中 NameNode 为 Server)、NameNode 和 DataNode(其中 DataNode 为 Client,NameNode 为 Server)、Client 和 DataNode(其中 DataNode 为 Server)、DataNode 和 DataNode(其中一个为 Client,另一个为 Server)之间的通信。通过 RPC 可以从网络上的计算机请求服务,而不

需要了解底层网络协议。

Hadoop 的 RPC 主要由 RPC Interface(对外编程接口)、Client(客户端实现)和 Server(服务器端实现)三部分组成。其中,RPC Interface 主要用于为通信的服务方提供代理;Client 主要用于连接 Server 端、传递函数参数名和相应的参数、等待结果等;Server 主要用于接受 Client 的请求、执行相应的函数、返回结果等,三者之间的交互过程如图 4.8 所示。

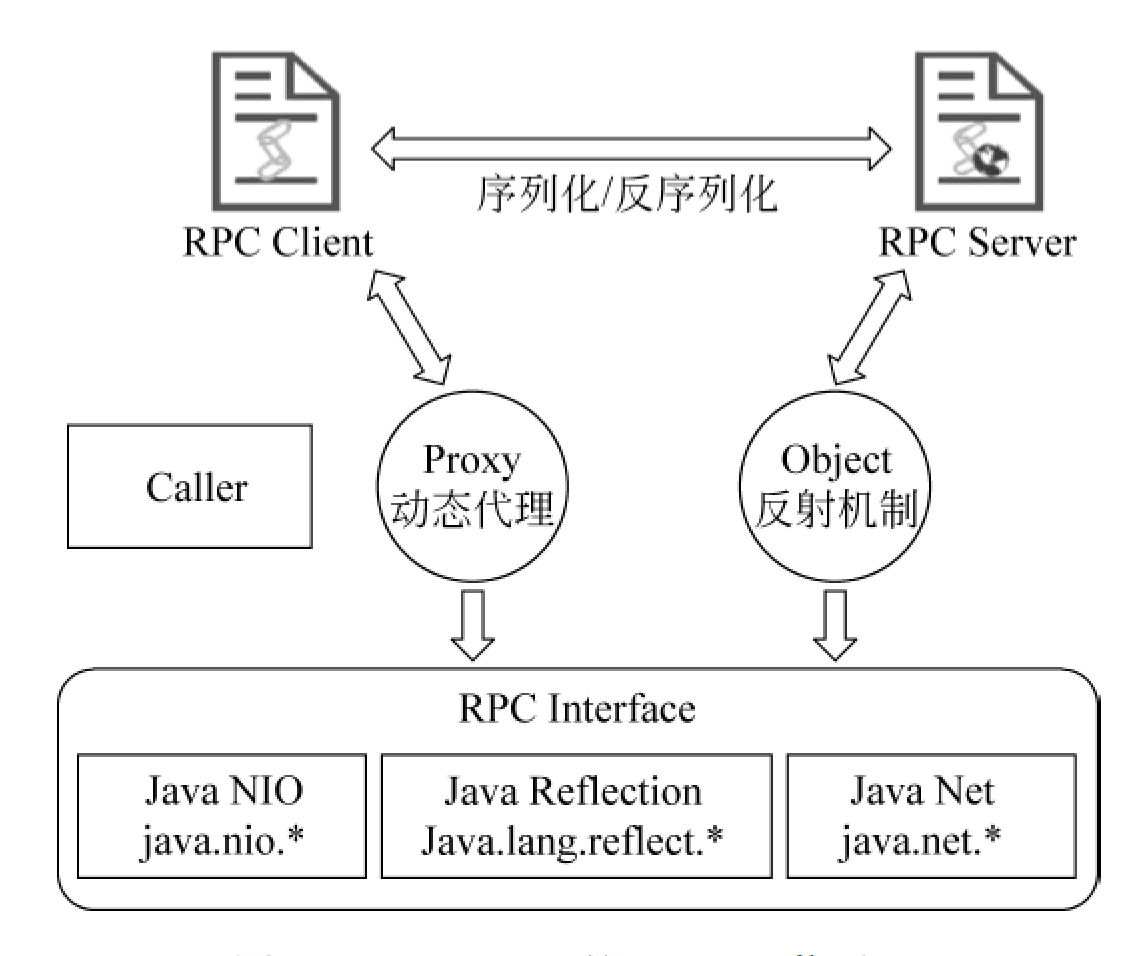


图 4.8 Hadoop 的 RPC 通信过程

从图 4.8 可以看出,Client 和 Server 之间在消息传输时,采用了 TCP/IP 的 Socket 机制,并将请求中的参数或者应答序列化/反序列化为字节流进行传输;在调用和执行函数时,是采用 Java 动态代理和反射机制来实现的;在具体实现时,RPC 用到了 JDK 提供的各种功能包,主要包括 java. nio. *(Java NIO 是 Java New IO 的简称,在 JDK 1.4 版本以上提供的新 API)、java. lang. reflect. *(反射机制和动态代理)和 java. net. *(网络编程库)等。

注:在 Hadoop 1.0 生态系统中,RPC 仅支持基于 Writable 序列化方式;在 Hadoop 2.0 生态系统中,RPC 提供了 Writable(WritableRpcEngine)和 Protocol Buffers(ProtobufRpcEngine)两种序列化框架,其默认实现是 Writable 方式,用户可通过调用 RPC. setProtocolEngine(…)修改采用的序列化方式。

4.5.1 RPC Interface

Hadoop 定义了自己的通信协议,实现了基于 IPC 模型的 RPC 机制。这些协议都是建立在 TCP/IP 之上,规范了通信两端的约定。在 Hadoop 1.0 生态系统中,存在一个 ipc 包,即 org. apache. hadoop. ipc 包。与 HDFS 有关的通信协议接口的继承关系如下所示。

```
org.apache.hadoop.ipc.VersionedProtocol
    |-org.apache.hadoop.hdfs.protocol.ClientProtocol
    |-org.apache.hadoop.hdfs.protocol.ClientDatanodeProtocol
```

```
|-org.apache.hadoop.hdfs.server.protocol.NamenodeProtocol
|-org.apache.hadoop.hdfs.server.protocol.DatanodeProtocol
|-org.apache.hadoop.hdfs.server.protocol.InterDatanodeProtocol
```

在 Hadoop 2.0 生态系统中存在两个 ipc 包,分别在 hadoop-common 项目中的 org.apache.hadoop.ipc 包和 hadoop-yarn-common 项目中的 org.apache.hadoop.yarn.ipc 包。其中,hadoop-common 项目的 ipc 包是 Hadoop RPC;hadoop-yarn-common 项目的 ipc 包是新的 YARN RPC。

1. VersionedProtocol

VersionedProtocol 是 RPC 协议接口的父接口。在 Hadoop 1.0 生态系统中,所有要使用 RPC 服务的类都要实现该接口;在 Hadoop 2.0 生态系统中,只有 ClientProtocol 继承了该接口。VersionedProtocol 接口的定义如下。

```
package org.apache.hadoop.ipc;
import java.io.IOException;

/**
 * 使用 Hadoop RPC 机制的所有协议的超类
 * 该接口的子类同样支持具有一个 static final long 的版本属性字段
 */
 public interface VersionedProtocol {

/**
 * 返回与指定协议 protocol 相关的协议版本
 * @param protocol 协议接口的类名
 * @param clientVersion 客户端欲与服务器进行交互,它所使用的协议版本
 * @return 返回服务器将要与客户端进行交互,所需要使用的协议版本
 */
 public long getProtocolVersion(String protocol, long clientVersion) throws IOException;

/**
 * Hadoop 2.0 生态系统中新加入的该方法,思考版本之间的兼容性问题
 * 返回与指定协议 protocol 相关的协议接口
 * @param protocol 协议接口的类名
 * @param clientVersion 客户端欲与服务器进行交互,它所使用的协议版本
 * @param clientMethodsHash 客户端协议方法的 hashcode
 */
 public ProtocolSignature getProtocolSignature (String protocol, long clientVersion, int
 clientMethodsHash) throws IOException;
}
```

2. ClientProtocol

该协议是用户进程(Client 进程或 DataNode 进程)与 NameNode 进程之间进行通信所使用的协议。当 Client 进程需要向 DataNode 数据节点复制数据块时,需要通过 ClientProtocol 协议与 NameNode 进程通信,从而获取 DataNode 节点列表;当 DataNode 进程向 NameNode 进程发送状态报告时,需要通过 ClientProtocol 协议与 NameNode 进程通信。该接口定义了对 HDFS 操作的很多方法,如打开、重命名、创建目录、删除、关闭文件流等 DFSClient 与 NameNode 交互的方法。如果要对 HDFS 上的文件进行操作,一般不会直接使用该接口,而是通过 org. apache. hadoop. fs. FileSystem 实现对 HDFS 的操作。该接口定义了一个 static final 的 versionID 字段,如下所示。

```
/**
 * 用来识别各个版本的 HDFS
 * 不同版本之间,无法进行通信
 * /
public static final long versionID= 61L;            //在 Hadoop 1.2.1 中,versionID 是 61L
public static final long versionID= 69L;            //在 Hadoop 2.6.0 中,versionID 是 69L
```

该接口协议中定义的与文件内容相关的操作的方法及说明如表 4.2 所示。

表 4.2 ClientProtocol 协议中常用方法

方法	说明
getBlockLocations()	获取指定文件的全部块信息,并返回一个对象 LocatedBlocks,该对象包括文件长度,组成文件的块及存储位置
create()	在文件系统命名中创建一个文件入口。一旦文件创建成功,可以被其他 Client 来执行读操作,但不能够对该文件进行删除、重命名、重写操作
append()	实现对现有文件的追加操作,只有当 HDFS 开启了 dfs. support. append 之后才可以进行文件的追加操作
setReplication()	为一个指定的文件修改块副本因子。如果当前副本因子小于设置的新副本因子,需要增加一些块副本,如果当前副本因子大于设置的新副本因子,就会删除一些副本
setPermission()	为已经存在的目录或者文件设置给定的操作权限
setOwner()	设置单个文件或者目录的 owner 信息和 group 信息
abandonBlock()	实现放弃对指定 Block 的写入操作
addBlock()	实现对已经打开的文件的写操作,返回值是可写 Block 的位置信息。如果文件比较大,要写多个 Block,那么就会多次调用该函数取得新 Block 的位置信息
complete()	对指定文件的写入操作完毕后调用该函数来完成操作
reportBadBlocks()	向 NameNode 报告已经损坏的 Block 信息

续表

方　　法	说　　明
rename()	实现 HDFS 中的文件更名操作
delete()	实现删除指定的文件或者目录
mkdirs()	实现创建相应名称和权限的文件夹
getListing()	实现查询指定文件夹的文件信息
renewLease()	实现文件加锁，防止其他的 Client 对其操作
getStats()	获得当前 HDFS 的信息，如总空间、已用空间、可用空间等
getDatanodeReport()	获得当前系统中 DataNode 的状态。其中，LIVE 表示 DataNode 存活，DEAD 表示 DataNode 已死
getPreferredBlockSize()	获得指定文件的块大小
setSafeMode()	实现进入或者离开 NameNode 的安全模式
refreshNodes()	从 NameNode 重新获取 hosts 信息和文件信息
finalizeUpgrade()	告诉 NameNode 完成更新
distributedUpgradeProgress()	获取当前 HDFS 的更新状态
metaSave()	将 NameNode 的数据结构信息写入到指定的文件中。如果文件已经存在，则进行追加操作
getFileInfo()	获得指定文件或者文件夹的信息
fsync()	把指定文件的元数据同步到物理存储上
setTimes()	设置文件的修改时间和访问时间

注： Hadoop 2.0 生态系统与 Hadoop 1.0 生态系统中 ClientProtocol 接口所定义的方法参数、返回类型等方面有所变化，请读者根据实际需求去理解。

3. ClientDatanodeProtocol

ClientDatanodeProtocol 协议是 Client 进程与 DataNode 进程之间进行通信所使用的协议。在 Hadoop 1.0 生态系统中（以 Hadoop 1.2.1 版本为例），ClientDatanodeProtocol 接口定义的方法如图 4.9 所示。

```
<<Interface>>
ClientDatanodeProtocol
versionID : long = 0 { readonly }
LOG : Log { readonly }
getBlockLocalPathInfo(block : ExtendedBlock, token : Token, ) : BlockLocalPathInfo
recoverBlock(block : Block, keepLength : boolean, targets : DatanodeInfo[*], ) : LocatedBlock
getBlockInfo(block : Block, ) : Block
```

图 4.9　Hadoop 1.0 生态系统中 ClientDatanodeProtocol 类图

从图 4.9 可以看出，ClientDatanodeProtocol 接口只定义了三个方法，即 recoverBlock()、getBlockInfo()和 getBlockLocalPathInfo()方法。其中，recoverBlock()方法用于恢复数据块；

getBlockInfo()和 getBlockLocalPathInfo()方法用于获得相应的块信息。在 Hadoop 2.0 生态系统中，由于 HDFS 的系统架构发生变化，因此，ClientDatanodeProtocol 接口的方法有所变化，如图 4.10 所示的 Hadoop 2.0 生态系统中 ClientDatanodeProtocol 类图(以 Hadoop 2.6.0 版本为例)。

```
<<interface>>
ClientDatanodeProtocol
versionID : long = 0 { readonly }
getBlockLocalPathInfo (block : ExtendedBlock, token : Token, ) : BlockLocalPathInfo
getReplicaVisibleLength (b : ExtendedBlock, ) : long
refreshNamenodes ()
deleteBlockPool (bpid : String, force : boolean)
getHdfsBlocksMetadata (blockPoolId : String, blockIds : long[*], tokens : List, ) : HdfsBlocksMetadata
shutdownDatanode (forUpgrade : boolean)
getDatanodeInfo () : DatanodeLocalInfo
startReconfiguration ()
getReconfigurationStatus () : ReconfigurationTaskStatus
```

图 4.10 Hadoop 2.0 生态系统中 ClientDatanodeProtocol 类图

从图 4.10 可以看出，在 Hadoop 2.0 生态系统中 ClientDatanodeProtocol 类提供了更为丰富的方法来获取相应 DataNode 的信息以及对存储块池(Block Pool)的操作。

4. NamenodeProtocol

NamenodeProtocol 协议用于 SecondaryNameNode 与 NameNode 之间的通信，并定义了 SecondaryNameNode 与 NameNode 之间进行通信所需的操作。其中，SecondaryNameNode 是一个用来辅助 NameNode 的服务器端进程，主要是对映像文件执行特定的操作，另外还包括获取指定 DataNode 上块的操作。该接口定义的方法如图 4.11 和图 4.12 所示。

```
<<interface>>
NamenodeProtocol
versionID : long = 0 { readonly }
getBlocks (datanode : DatanodeInfo, size : long, ) : BlocksWithLocations
getBlockKeys () : ExportedBlockKeys
getEditLogSize () : long
rollEditLog () : CheckpointSignature
rollFsImage ()
```

图 4.11 Hadoop 1.0 生态系统中 NamenodeProtocol 类图

其中，getBlocks()方法用于获取 DataNode 上大小为 size 的块；getEditLogSize()方法用于获取 EditLog 文件的大小；rollEditLog()方法用于关闭当前 EditLog 文件，并重新打开一个新的 EditLog 文件；rollFsImage()方法用于回滚 FsImage 日志。

5. DatanodeProtocol

DatanodeProtocol 协议用于 DataNode 与 NameNode 之间的通信，并定义了 DataNode 与 NameNode 之间进行通信所需的操作，如发送心跳报告和块状态报告等。该接口定义的方法(以 Hadoop 2.6.0 版本为例)如下所示。

```
<<interface>>
NamenodeProtocol
versionID : long = 0 { readonly }
NOTIFY : int = 0 { readonly }
FATAL : int = 1 { readonly }
ACT_UNKNOWN : int = 0 { readonly }
ACT_SHUTDOWN : int = 50 { readonly }
ACT_CHECKPOINT : int = 51 { readonly }
getBlocks(datanode : DatanodeInfo, size : long, ) : BlocksWithLocations
getBlockKeys() : ExportedBlockKeys
getTransactionID() : long
getMostRecentCheckpointTxId() : long
rollEditLog() : CheckpointSignature
versionRequest() : NamespaceInfo
errorReport(registration : NamenodeRegistration, errorCode : int, msg : String)
registerSubordinateNamenode(registration : NamenodeRegistration, ) : NamenodeRegistration
startCheckpoint(registration : NamenodeRegistration, ) : NamenodeCommand
endCheckpoint(registration : NamenodeRegistration, sig : CheckpointSignature)
getEditLogManifest(sinceTxId : long, ) : RemoteEditLogManifest
```

图 4.12　Hadoop 2.0 生态系统中 NamenodeProtocol 类图

```
public interface DatanodeProtocol {
  //DatanodeProtocol 版本号
  public static final long versionID= 28L;
  //定义错误代码
  final static int NOTIFY= 0;
  final static int DISK_ERROR= 1;                 //there are still valid volumes on DN
  final static int INVALID_BLOCK= 2;
  final static int FATAL_DISK_ERROR= 3;           //no valid volumes left on DN

  /**
   * 当接收到 DataNode 的命令时,根据下述状态码确定 DataNode 应该执行何种操作
   * /
  final static int DNA_UNKNOWN= 0;                //未知
  final static int DNA_TRANSFER= 1;               //将数据块转移到另一个 DataNode
  final static int DNA_INVALIDATE= 2;             //未验证数据块
  final static int DNA_SHUTDOWN= 3;               //关闭 DataNode
  final static int DNA_REGISTER= 4;               //重新注册
  final static int DNA_FINALIZE= 5;               //完成先前执行的升级操作
  final static int DNA_RECOVERBLOCK= 6;           //数据块恢复操作请求
  final static int DNA_ACCESSKEYUPDATE= 7;        //更新 access key
  final static int DNA_BALANCERBANDWIDTHUPDATE= 8;        //更改带宽
  final static int DNA_CACHE= 9;                  //缓存块
  final static int DNA_UNCACHE= 10;               //非缓存块
  //注册 Datanode.
  public  DatanodeRegistration  registerDatanode  ( DatanodeRegistration  registration )  throws
  IOException;
  //DataNode 向 NameNode 发送心跳状态报告
  public HeartbeatResponse sendHeartbeat(DatanodeRegistration registration,
                                         StorageReport[] reports,
```

```
                                             long dnCacheCapacity,
                                             long dnCacheUsed,
                                             int xmitsInProgress,
                                             int xceiverCount,
                                             int failedVolumes)throws IOException;
  //DataNode 向 NameNode 发送块状态报告
  public DatanodeCommand blockReport (DatanodeRegistration registration, String poolId,
  StorageBlockReport[] reports)throws IOException;
  //DataNode 向 NameNode 发送 cache 状态报告
  public DatanodeCommand cacheReport(DatanodeRegistration registration,String poolId, List<Long>
 blockIds)throws IOException;
  //DataNode 向 NameNode 发送已经接收或删除 Block 信息
  public void blockReceivedAndDeleted(DatanodeRegistration registration,
                            String poolId,
                            StorageReceivedDeletedBlocks[] rcvdAndDeletedBlocks)
                                  throws IOException;
  //向 NameNode 发送报告错误信息
  public void errorReport(DatanodeRegistration registration,
                                int errorCode,
                                String msg)throws IOException;
  //向 NameNode 发送版本请求信息
  public NamespaceInfo versionRequest()throws IOException;
  //DataNode 向 NameNode 报告 Bad Blocks
  public void reportBadBlocks(LocatedBlock[] blocks)throws IOException;
  //在恢复数据块期间,提交事务,数据块同步
  public void commitBlockSynchronization(ExtendedBlock block,
       long newgenerationstamp, long newlength,
       boolean closeFile, boolean deleteblock, DatanodeID[] newtargets,
       String[] newtargetstorages)throws IOException;
}
```

6. InterDatanodeProtocol

InterDatanodeProtocol 协议用于 DataNode 与 DataNode 之间的通信，并定义了 DataNode 与 DataNode 之间进行通信所需的操作，如 DataNode 节点之间块副本的复制操作。该接口定义的方法如图 4.13(以 Hadoop 1.2.1 版本为例)和图 4.14(以 Hadoop 2.6.0 版本为例)所示。

从图 4.13 可以看出，在 Hadoop 1.0 生态系统中 InterDatanodeProtocol 类提供了三个方法，主要功能是获取指定块的元数据，并更新数据块。

从图 4.14 可以看出，在 Hadoop 2.0 生态系统中 InterDatanodeProtocol 类提供了两个方法，主要功能是初始化要恢复的数据块，并更新数据块。虽然 Hadoop 2.0 生态系统中 InterDatanodeProtocol 接口提供的方法与 Hadoop 1.0 生态系统中 InterDatanodeProtocol 接




```
<<interface>>
InterDatanodeProtocol
(from protocol)
LOG : Log { readonly }
versionID : long = 0 { readonly }
getBlockMetaDataInfo(block : Block, ) : BlockMetaDataInfo
startBlockRecovery(block : Block, ) : BlockRecoveryInfo
updateBlock(oldblock : Block, newblock : Block, finalize : boolean)
```

图 4.13　Hadoop 1.0 生态系统中 InterDatanodeProtocol 类图

```
<<interface>>
InterDatanodeProtocol
(from protocol)
LOG : Log { readonly }
versionID : long = 0 { readonly }
initReplicaRecovery(rBlock : RecoveringBlock, ) : ReplicaRecoveryInfo
updateReplicaUnderRecovery(oldBlock : ExtendedBlock, recoveryId : long, newLength : long, ) : String
```

图 4.14　Hadoop 2.0 生态系统中 InterDatanodeProtocol 类图

口提供的方法有所不同,但实现的功能是完全相同的。

上面对不同进程之间通信所使用的协议的接口进行了分析,请读者了解每种协议的应用场景。如果想要基于某种场景实现 RPC 通信,可以选择合适的协议接口来实现这些接口。

4.5.2　RPC Client

RPC Client 通过 Socket 将调用的业务方法和参数传送至 Server 端,并等待 Server 端的响应。RPC Client 在 Hadoop 中的实现为一个类:org. apache. hadoop. ipc. Client,该类使用 Java 的动态代理技术,生成 Server 端的业务接口的代理。RPC Client 端的 RPC 调用处理过程如图 4.15 所示。

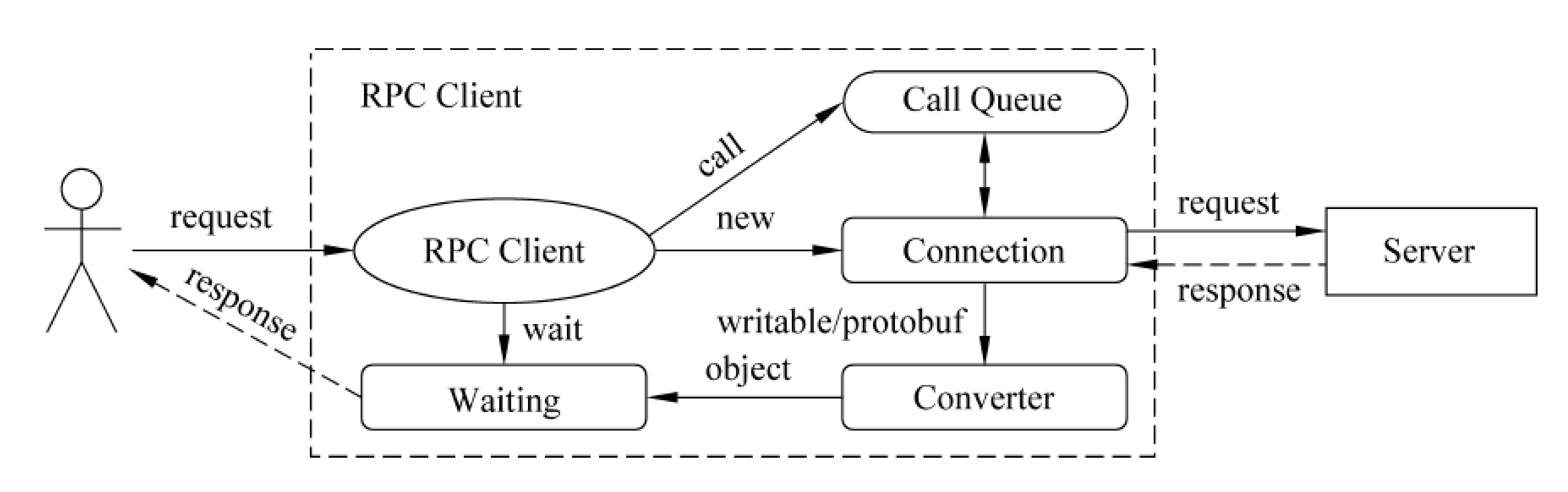


图 4.15　Client 端 RPC 调用处理过程

从图 4.15 可以看出,Client 端 RPC 调用的处理过程由一系列实体组成,各实体分工明确。Client 首先创建一个通向 Server 端的连接 Connection,并用 Call 封装好调用信息放入 Call Queue 中,这样 Client 就可以同时发生很多调用,每个调用用 ID 来识别。然后,Client 通过 Socket 将 Connection 序列化后的调用信息(调用信息是 Client 的调用方,如 NameNode 或 DataNode 指定)发送到 Server 端,并等待结果。最后,Connection 读取 Server 端的返回信息,等待 Connection 接收完响应返回。

RPC Client 类提供的最基本的功能就是进行 RPC 调用，是 RPC Client 端的实现和入口类。在 Hadoop 1.0 生态系统中，该类定义了 5 个内部类来完成该功能，即 Client. Call、Client. ConnectionId、Client. ParalelResults、Client. Connection 和 Client. ParallelCall 类。其中，Client. Call 是 Client 端调用的一个抽象，定义了一次调用所需要的条件，以及修改 Client 端的一些全局统计变量，主要用于存储 Call 调用信息；Client. ConnectionId 是一个连接的实体类，标识了一个连接实例的 Socket 地址、用户信息和连接协议类，主要用于标识到 RPC Server 端的连接对象；Client. ParallelResults 类是用来收集在并行调用环境中结果的实体类，主要用于存储响应信息；Client. ParallelCall 类继承于内部类 Call，返回值使用 ParallelResults 实体类来封装；Client. Connection 类是一个连接管理内部的线程类，继承自 Thread 类，主要用于接收数据、解析数据包。在 Hadoop 2.0 生态系统中，RPC Client 类内部定义了 4 个内部类来完成 RPC 调用功能，即 Client. Call、Client. ClientExecutorServiceFactory、Client. Connection 和 ConnectionId 类（具体实现细节请查看源码）。

4.5.3 RPC Server

RPC Server 通过 Socket 监听 Client 端的请求，获取 Client 端需要调用的方法和参数之后，使用 Java 反射机制来调用相对应的方法，并且将结果返回到 Client 端。RPC Server 在 Hadoop 中的实现为一个类：org. apache. hadoop. ipc. Server。RPC Server 端的 RPC 调用处理过程如图 4.16 所示。

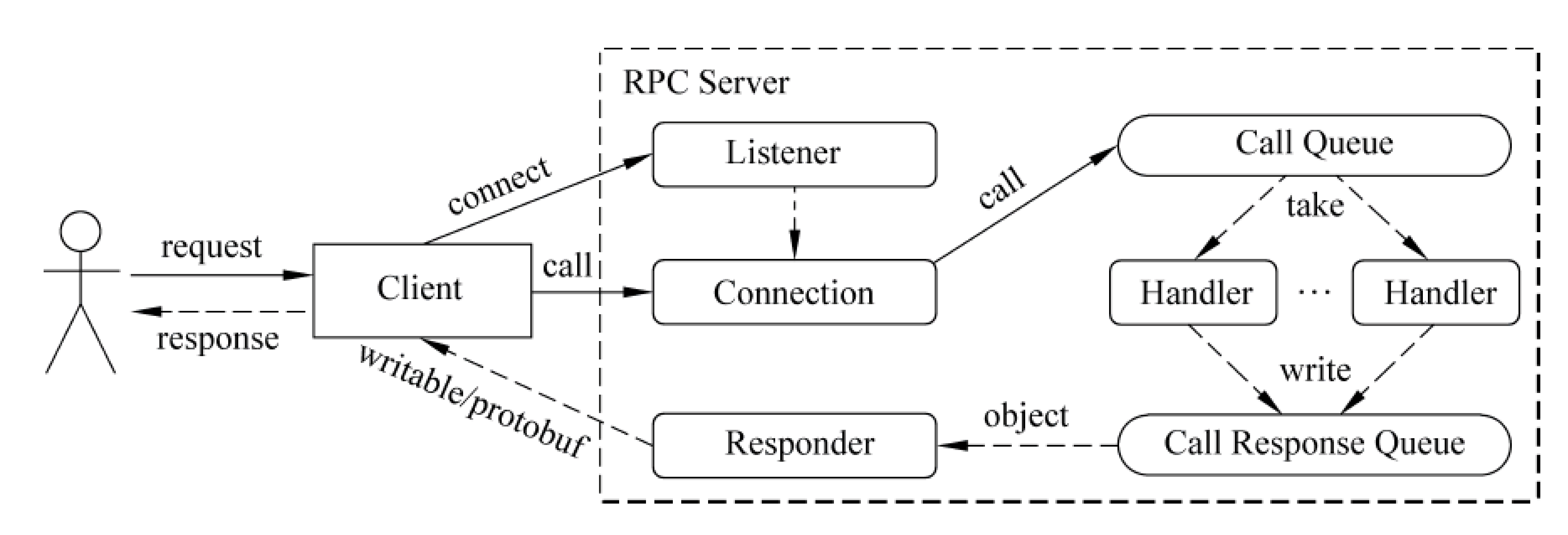


图 4.16　Server 端 RPC 调用处理过程

从图 4.16 可以看出，首先 Client 发送 TCP 连接，启动 Listener 进程，用来接收 RPC Client 的连接请求和数据，如果收到需要建立连接的请求，将建立连接并捕获读操作的命令。然后，Listener 进程收到读操作命令后，把 Client 端发过来的调用信息委派给 Connection。Connection 把调用信息封闭到 Call 对象中，并放入 Call Queue 队列中，等待 Handler 处理。最后，Server 启动指定数目的 Handler 线程，处理来自 Client 端对指定方法调用的请求，并把结果写入到 Call Response Queue 队列中，并调用 Responder 将结果写入 Client 端。

RPC Server 类是 Server 端的抽象实现，定义了一个抽象的 IPC 服务。该 IPC 服务接收 Client 端发送的参数值，并返回响应值。在 Hadoop 1.0 生态系统中，Server 类定义了 6 个内部类，即 Server. Call、Server. Connection、Server. Listener、Server. Handler、

ExceptionsHandler 和 Server. Responder 类。其中，Server. Call 类是 Server 端使用队列维护的调用实体类，其属性包括 Client 端调用 Call 的 ID、Client 端调用用传递的参数 param、Client 端的连接实例 connection 等信息，主要用于存储 Client 端的 Call 信息；Server. Connection 维护了一个来自 Client 端的 Socket 连接，主要用于接收数据、解析数据包；Server. Listener 类继承自 Thread 线程类，主要用于接收 Client 端的连接请求，并为 Handler 创建处理任务；Server. Handler 类是一个处理线程类，从 Call Queue 队列中获取调用信息，然后反射调用真正的对象并得到结果，再把此次调用放到响应队列 Call Response Queue 里；Server. Responder 类是实现发送 RPC 响应到 Client 端，它不断地检查响应队列中是否有调用信息，如果有调用信息，就把调用的结果返回给 Client 端。在 Hadoop 2.0 生态系统中，Server 类定义了 9 个内部类，即 Server. WrappedRpcServerException、Server. Listener、Server. Handler、Server. Call、Server. Connection、Server. ConnectionManager、ExceptionsHandler、Server. RpcKindMapValue 和 Server. Responder 类，其 Server 核心功能的实现主要依赖于 Listener、Handler、Call、Connection 和 Responder 类（具体实现细节请查看源码）。

总而言之，Hadoop 1.0 生态系统中的 RPC 结构比较简单，实现类也只有 6 个，而 Hadoop 2.0 生态系统中的 RPC 相对来说比较复杂，实现了三十多个类，最主要的改变是序列化与反序列化不再采用 Writable 接口，而是采用 PB(Protocol Buffer)，底层的动态代理(Dynamic Proxy)以及 TCP 连接等变化不大。

4.5.4 RPC 通信实现

Hadoop 中的 org. apache. hadoop. ipc. RPC 类利用 Java 的动态代理(Dynamic Proxy)与反射机制(Reflect)实现了 RPC 通信。其中，动态代理类是由 java. lang. reflect. Field、java. lang. reflect. Proxy 和 java. lang. reflect. InvocationHandler 类在运行期间根据接口，采用 Java 反射机制实现的；生成的动态代理类并不负责实际的处理工作，只负责将该类上的所有方法的调用传递到 java. lang. reflect. InvocationHandler，由 InvocationHandler 来处理 Client 端的请求。org. apache. hadoop. ipc. RPC 类是对 Server 和 Client 的具体化，并对外主要提供了两种接口，即 getProxy/waitForProxy()和 RPC. Builder. build()。其中，getProxy/waitForProxy()用于构造一个 Client 端代理对象，并向 Server 端发送 RPC 请求，如果要销毁 Client 端代理，只提供了一个方法，即 stopProxy；RPC. Builder. build()用于构造一个 Server 端对象，用于处理 Client 端发送来的请求。

以 Hadoop 2.0 生态系统为例(Hadoop 2.6.0 版本)，介绍 Client 端与 Server 端之间的 RPC 通信的代码实现。要实现 Client 端的动态代理就需要调用 PRC 类的 getProxy()，该方法最终调用 Java 动态代理的方法创建一个代理类，并创建了 InvocationHandler 的实例。PRC 类的 getProxy()方法代码实现如下。

```
public static <T> T getProxy(Class<T> protocol,
                             long clientVersion,
```

```
                    InetSocketAddress addr, Configuration conf,
                              SocketFactory factory) throws IOException {
    return getProtocolProxy (
        protocol, clientVersion, addr, conf, factory).getProxy();
  }
```

通过 getProxy()创建的 InvocationHandler 实例用于 Client 端向 Server 端发送 RPC 请求以及获取响应。该实例是由 RPC 类中的 RpcInvoker 实现的,其代码实现如下。

```
interface RpcInvoker {
    /**
     * Process a client call on the server side
     * @param server the server within whose context this rpc call is made
     * @param protocol- the protocol name (the class of the client proxy
     *       used to make calls to the rpc server.
     * @param rpcRequest  - deserialized
     * @param receiveTime time at which the call received (for metrics)
     * @return the call's return
     * @throws IOException
     **/
    public Writable call(Server server, String protocol,
        Writable rpcRequest, long receiveTime) throws Exception ;
  }
```

RPC Server 端的构建则由 RPC 静态内部类 RPC.Builder 实现,该类提供了一些 set ***方法供用户设置一些基本参数,如 RPC 协议、RPC 协议实现对象等。当设置完这些参数后,可通过调用 RPC.Builder.build()方法完成 Server 端对象的创建。RPC.Builder 代码实现如下。

```
public static class Builder {
    private Class<?> protocol= null;
    private Object instance= null;
    private String bindAddress= "0.0.0.0";
    private int port= 0;
    private int numHandlers= 1;
    private int numReaders= - 1;
    private int queueSizePerHandler= - 1;
    private boolean verbose= false;
    private final Configuration conf;
    private SecretManager<?extends TokenIdentifier> secretManager= null;
    private String portRangeConfig= null;

    public Builder(Configuration conf){
```

```
        this.conf = conf;
    }

    /** Mandatory field */
    public Builder setProtocol(Class<?> protocol) {
        this.protocol = protocol;
        return this;
    }

    /** Mandatory field */
    public Builder setInstance(Object instance) {
        this.instance = instance;
        return this;
    }

    /** Default: 0.0.0.0 */
    public Builder setBindAddress(String bindAddress) {
        this.bindAddress = bindAddress;
        return this;
    }

    /** Default: 0 */
    public Builder setPort(int port) {
        this.port = port;
        return this;
    }

    /** Default: 1 */
    public Builder setNumHandlers(int numHandlers) {
        this.numHandlers = numHandlers;
        return this;
    }

    /** Default: -1 */
    public Builder setnumReaders(int numReaders) {
        this.numReaders = numReaders;
        return this;
    }

    /** Default: -1 */
    public Builder setQueueSizePerHandler(int queueSizePerHandler) {
        this.queueSizePerHandler = queueSizePerHandler;
        return this;
```

```
    }

    /** Default: false */
    public Builder setVerbose(boolean verbose) {
      this.verbose=verbose;
      return this;
    }

    /** Default: null */
    public Builder setSecretManager(
        SecretManager< ?extends TokenIdentifier> secretManager) {
      this.secretManager=secretManager;
      return this;
    }

    /** Default: null */
    public Builder setPortRangeConfig(String portRangeConfig) {
      this.portRangeConfig=portRangeConfig;
      return this;
    }

    /**
     * Build the RPC Server.
     */
    public Server build() throws IOException, HadoopIllegalArgumentException {
      if(this.conf==null) {
    throw new HadoopIllegalArgumentException("conf is not set");
      }
      if(this.protocol==null) {
  throw new HadoopIllegalArgumentException("protocol is not set");
      }
      if(this.instance==null) {
  throw new HadoopIllegalArgumentException("instance is not set");
      }

      return getProtocolEngine(this.protocol,this.conf).getServer(
          this.protocol, this.instance, this.bindAddress,this.port,
          this.numHandlers, this.numReaders, this.queueSizePerHandler,this.verbose, this.conf,
          this.secretManager, this.portRangeConfig);
    }
  }
```

RPC.Builder 创建了一个 RPC Server，而 RPC Server 的具体实现是由 PRC 的内部

静态类 Server 继承 org. apache. hadoop. ipc. Server 类实现的(RPC. Server 的具体实现代码请查看 org. apache. hadoop. ipc. RPC 类中的静态类 Server)。

当不需要 Client 代理时，可以使用 RPC 类中的 stopProxy()方法取消代理。stopProxy()方法的实现代码如下。

```
public static void stopProxy(Object proxy) {
    if (proxy==null) {
      throw new HadoopIllegalArgumentException(
          "Cannot close proxy since it is null");
    }
    try {
      if (proxy instanceof Closeable) {
        ((Closeable)proxy).close();
        return;
      } else {
        InvocationHandler handler= Proxy.getInvocationHandler(proxy);
        if (handler instanceof Closeable) {
          ((Closeable)handler).close();
          return;
        }
      }
    } catch (IOException e) {
      LOG.error("Closing proxy or invocation handler caused exception", e);
    } catch (IllegalArgumentException e) {
      LOG.error("RPC.stopProxy called on non proxy: class="+proxy.getClass().getName(), e);
    }

    throw new HadoopIllegalArgumentException(
        "Cannot close proxy- is not Closeable or "
            +"does not provide closeable invocation handler "
            +proxy.getClass());
  }
```

4.6　HDFS 安全机制

系统的安全机制一般由认证(Authentication)和授权(Authorization)两部分组成。在 Hadoop 1.0 生态系统中，最初版本的 Hadoop 并没有安全机制可言。虽然在 Hadoop 0.16 版本后，HDFS 增加了文件和目录权限，但是并没有实现强认证，从而导致恶意用户伪装成真正的用户入侵到 Hadoop 集群篡改文件和目录权限，篡改 HDFS 上的数据，以及伪装 NameNode 等。在 Hadoop 2.0 生态系统中，Hadoop 提供了两种认证机制和一种授权机制。下面将以 Hadoop 2.0 生态系统为例(以 Hadoop 2.6.0 stable 版本为例)，分别对 Hadoop 所提供的授权机制和认证机制进行讲解。

4.6.1 授权机制

Hadoop 的授权机制是通过引入访问控制列表(Access Control List,ACL)实现的。Hadoop 通过制定接口协议的方式来实现节点之间服务调用的逻辑,每个协议指定的一组服务,再基于用户组来确定是否有权限执行某一种协议所包含的集合。如果需要启动该授权方式,首先需要在 Hadoop 的配置文件 $HADOOP_HOME/etc/Hadoop/core-site. xml 中增加以下配置内容。

```
<property>
  <name>hadoop.security.authorization</name>
  <value>true</value>
  <description>Is service-level authorization enabled?</description>
</property>
```

其中,hadoop. security. authorization 默认值为 false,将 false 改为 true。设置完成后,需要将 core-site. xml 同步到集群中的所有机器,并重新启动 Hadoop 集群。

然后,需要在配置文件 $HADOOP_HOME/etc/Hadoop/hadoop-policy. xml 中,对 ACL 进行详细配置。该配置文件 hadoop-policy. xml 可以实现对 HDFS、MapReduce 和 YARN 进行 ACL 配置,其中与 HDFS 有关的配置项和协议之间的对应关系如表 4. 3 所示。

表 4. 3 配置项与协议之间的对应关系

配置项	协议名
security. client. protocol. acl	ClientProtocol
security. client. datanode. protocol. acl	ClientDatanodeProtocol
security. datanode. protocol. acl	DatanodeProtocol
security. inter. datanode. protocol. acl	InterDatanodeProtocol
security. namenode. protocol. acl	NamenodeProtocol
security. admin. operations. protocol. acl	AdminOperationsProtocol
security. refresh. user. mappings. protocol. acl	RefreshUserMappingsProtocol
security. refresh. policy. protocol. acl	RefreshAuthorizationPolicyProtocol
security. ha. service. protocol. acl	HAServiceProtocol
security. zkfc. protocol. acl	ZKFailoverController
security. qjournal. service. protocol. acl	QJournalProtocol
security. mrhs. client. protocol. acl	HSClientProtocol

其中,ClientProtocol 是用户进程与 NameNode 交互协议,用来操作 Namespace,以及打开/关闭文件流等;ClientDatanodeProtocol 是 Client 端与 DataNode 交互协议,用来实现数据恢复;DatanodeProtocol 是 DataNode 与 NameNode 之间的通信协议,用来实现 DataNode 和 NameNode 之间所需的操作;InterDatanodeProtocol 是 DataNode 之间进行通信的协议,用来更新 Block 副本;NamenodeProtocol 是 SecondaryNameNode 与 NameNode 进行

通信的协议，用来获取 NameNode 的状态信息；AdminOperationsProtocol 是 HDFS 管理操作协议；RefreshUserMappingsProtocol 用来刷新缓存中用户与用户组映射关系信息；RefreshAuthorizationPolicyProtocol 用来更新认证策略(Authorization Policy)配置，对应于配置文件/etc/hadoop/hadoop-policy.xml，控制执行 hdfs dfsadmin -refreshServiceAcl 和 yarn rmadmin -refreshServiceAcl 的权限；HAServiceProtocol 是 HDFS HA 操作协议，用来管理 Active NameNode 与 Stand-by NameNode 状态；ZKFailoverController 是 ZooKeeper Failover 控制器操作权限协议，用于 HDFS HA；QJournalProtocol 是 QuorumJournalManager 与 JournalNode 之间通信的协议，用于 HDFS HA，用来同步 edits，并协调 Active NameNode 与 Stand-by NameNode 状态；HSClientProtocol 是 Client 端与 MR History Server 之间通信的协议，用来查看 Job 历史信息。

注：由于不同 Hadoop 版本的配置文件 hadoop-policy.xml 中的配置项有所不同，为了方便读者理解，本实例是以 Hadoop 2.6.0 版本为例。

在本实例的配置文件 hadoop-policy.xml 中与 HDFS 有关的默认配置内容如下。

```
<property>
  <name>security.client.protocol.acl</name>
  <value>*</value>
</property>

<property>
  <name>security.client.datanode.protocol.acl</name>
  <value>*</value>
</property>

<property>
  <name>security.datanode.protocol.acl</name>
  <value>*</value>
</property>

<property>
  <name>security.inter.datanode.protocol.acl</name>
  <value>*</value>
</property>

<property>
  <name>security.namenode.protocol.acl</name>
  <value>*</value>
</property>

<property>
  <name>security.admin.operations.protocol.acl</name>
```

```
    <value> * </value>
  </property>

  <property>
    <name>security.refresh.user.mappings.protocol.acl</name>
    <value> * </value>
  </property>

  <property>
    <name>security.refresh.policy.protocol.acl</name>
    <value> * </value>
  </property>

  <property>
    <name>security.ha.service.protocol.acl</name>
    <value> * </value>
  </property>

  <property>
    <name>security.zkfc.protocol.acl</name>
    <value> * </value>
  </property>

  <property>
    <name>security.qjournal.service.protocol.acl</name>
    <value> * </value>
  </property>

  <property>
    <name>security.mrhs.client.protocol.acl</name>
    <value> * </value>
  </property>
```

其中,<value> * </value>表示所有用户都具有对应的服务操作权限。如果要实现不同用户或用户组具有不同的授权时,就需要根据实际需求进行配置。为了方便读者理解,下面举几个示例来说明。

如果只允许 hadoop 用户具有修改 ACL 授权权限的配置,可以配置如下。

```
  <property>
    <name>security.refresh.policy.protocol.acl</name>
    <value>hadoop</value>
  </property>
```

如果只运行属于某个组内的所有用户(假定为 datanodes_users 用户组)可以运行 DataNode 和 NameNode 进行通信,可以配置如下。

```
<property>
  <name>security.datanode.protocol.acl</name>
  <value>[空格] datanodes_users</value>
</property>
```

如果只允许某个组内的某个用户(假定为 hdfs_client 用户组下的 hdfs_user1 用户)具有操作 HDFS 权限,可以配置如下。

```
<configuration>
  <property>
    <name>security.client.protocol.acl</name>
    <value>hdfs_user1 </value>
  </property>
```

如果允许所有的用户访问集群上的 HDFS,可以配置如下。

```
<configuration>
  <property>
    <name>security.client.protocol.acl</name>
    <value> * </value>
  </property>
```

为了保证 Hadoop 集群的配置相同,需要将修改后的配置文件同步到整个集群的所有节点上。如果修改了有关 NameNode 的服务配置,可以使用下面的命令来动态加载。

```
hdfs dfsadmin -refreshServiceAcl
```

注:在配置 hadoop-policy.xml 文件时需要注意,如果配置项中既有用户,又有用户组,配置内容格式为:user1、user2 group1、group2;如果只有用户组,配置内容前面需要增加一个空格,如:[空格]group1、group2。

4.6.2　认证机制

在 HDFS 中,当用户需要读写 Hadoop 集群上的文件时,需要实现 NameNode 认证;当 Client 需要读写 DataNode 数据时,同样也需要 DataNode 的认证;当 DataNode 与 NameNode 进行交互以及 SecondNameNode 与 NameNode 进行交互时,都需要 NameNode 的认证。在 Hadoop 2.0 生态系统中,Hadoop 给出了两种认证机制:Simple 机制和 Kerberos 机制。其中,Simple 机制是一种 JAAS 协议与 Delegation Token 整合机制;Kerberos 机制用于 Client 与 NameNode 初次通信以及 DataNode 与 NameNode 之间的认证。

1. Simple 机制

Simple 机制是 JAAS 协议(Java Authentication and Authorization Service,Java 认证和授权服务)与 Delegation Token 整合机制,Hadoop 默认采用 Simple 机制。该机制的配置可以通过修改配置文件 $HADOOP_HOME/etc/Hadoop/core-site. xml 中的 hadoop. security. authentication 参数进行设置。有关配置文件 core-site. xml 中 hadoop. security. authentication 参数的默认配置如下所示。

```
<property>
  <name>hadoop.security.authentication</name>
  <value>simple</value>
  <description>Possible values are simple(no authentication), and kerberos
  </description>
</property>
```

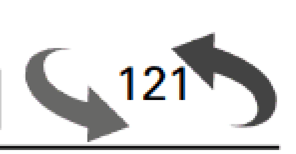
如在一个 HDFS 读操作过程中,当 Client 与 NameNode 之间第一次 RPC 调用时,并没有 Delegation Token 生成,所以需要 Kerberos 认证。一旦通过认证,Client 将会从 NameNode 中获得一个 Delegation Token(访问不同模块有不同的 Delegation Token)。之后的任何操作都将采用 Simple 机制。比如访问文件,均要检查该 Token 是否存在,且使用者跟之前注册使用该 Token 的用户是否一致,其具体过程为:Client 会使用一个特殊的 Delegation Token,叫做 Block Access Token。Client 使用这个 Token 来向 DataNode 认证自己(NameNode 和 DataNode 之间共享这个 Token),如果认证成功,该 Block 就能被持有这个 Token 的 Client 端进行访问了。

目前常用的 Simple 机制认证的方式有以下两种。

(1) 用户名/密码认证。这是一种常用方式,当用户数量比较多时,可以采用 SQL/LDAP 认证方式。这种方式通常性能较好。

(2) 机器地址过滤。这种方式通常有两种具体的实现,即黑名单列表和白名单列表。黑名单列表:机器不能访问;白名单列表:机器能够访问。这种方式尽管配置简单,但是部署比较麻烦,通常表现在:一次修改到处部署。Hadoop 通过配置 hdfs-site. xml 文件中的配置参数:dfs. hosts(允许访问 NameNode)和 dfs. hosts. exclude(不允许访问 NameNode)实现此种方式。但 Hadoop 默认并不使用,其默认配置如下。

```
<property>
  <name>dfs.hosts</name>
  <value></value>
  <description>Names a file that contains a list of hosts that are
  permitted to connect to the namenode. The full pathname of the file
  must be specified.  If the value is empty, all hosts are
  permitted.</description>
</property>
```




```
<property>
  <name>dfs.hosts.exclude</name>
  <value></value>
  <description>Names a file that contains a list of hosts that are
  not permitted to connect to the namenode.  The full pathname of the
  file must be specified.  If the value is empty, no hosts are
  excluded.</description>
</property>
```

2. Kerberos 机制

Kerberos 认证机制是一个基于共享密钥对称加密的安全网络认证系统，它避免了将密码在网上传输，而是将密码作为对称加密的密钥，通过能不能解密来验证用户的身份。在 Hadoop 中，Kerberos 认证机制默认是关闭的，可将配置文件 core-site. xml 中的 hadoop. security. authentication 参数设为“kerberos”(默认值为“simple”)来启动它，其配置内容如下。

```
<property>
  <name>hadoop.security.authentication</name>
  <value>kerberos</value>
  <description>Possible values are simple(no authentication), and kerberos
  </description>
</property>
```

要为 HDFS 添加 Kerberos 认证机制，除需要修改 hadoop. security. authentication 的配置参数外，还需要修改该文件中的其他配置参数、hdfs-site. xml 中的相应参数以及“/etc/krb5. conf”等配置。其中，core-site. xml 和 hdfs-site. xml 中需要修改的配置参数如表 4. 4 所示。

表 4. 4　配置项及参数值

配　置　项	参　考　值	配 置 文 件
hadoop. security. authorization	true	core-site. xml
hadoop. security. authentication	kerberos	core-site. xml
dfs. block. access. token. enable	true	hdfs-site. xml
dfs. namenode. keytab. file	/xx/xx/hadoop. keytab	hdfs-site. xml
dfs. namenode. kerberos. principal	hadoop/_HOST@HADOOP. COM	hdfs-site. xml
dfs. namenode. kerberos. internal. spnego. principal	HTTP/_HOST@HADOOP. COM	hdfs-site. xml
dfs. datanode. keytab. file	/xx/xx/hadoop. keytab	hdfs-site. xml

续表

配　置　项	参　考　值	配 置 文 件
dfs. datanode. kerberos. principal	hadoop/_HOST@HADOOP. COM	hdfs-site. xml
dfs. datanode. address	1004（小于 1024）	hdfs-site. xml
dfs. datanode. http. address	1006（小于 1024）	hdfs-site. xml
dfs. journalnode. keytab. file	/xx/xx/hadoop. keytab	hdfs-site. xml
dfs. journalnode. kerberos. principal	hadoop/_HOST@HADOOP. COM	hdfs-site. xml
dfs. journalnode. kerberos. internal. spnego. principal	HTTP/_HOST@HADOOP. COM	hdfs-site. xml

注：表 4.4 的配置中 keytab(包含 host 和对应节点的名字，以及它们之间的密钥)文件使用绝对路径，principal(被认证的实体，包含一个名字和口令)使用_HOST，Hadoop 会自动替换为对应的域名。

Kerberos 认证机制至少需要有三个角色：认证服务器(Authentication Server，AS)、Kerberos 客户端(Client)和 Kerberos 服务器(Server)。Kerberos 的认证机制如图 4.17 所示。

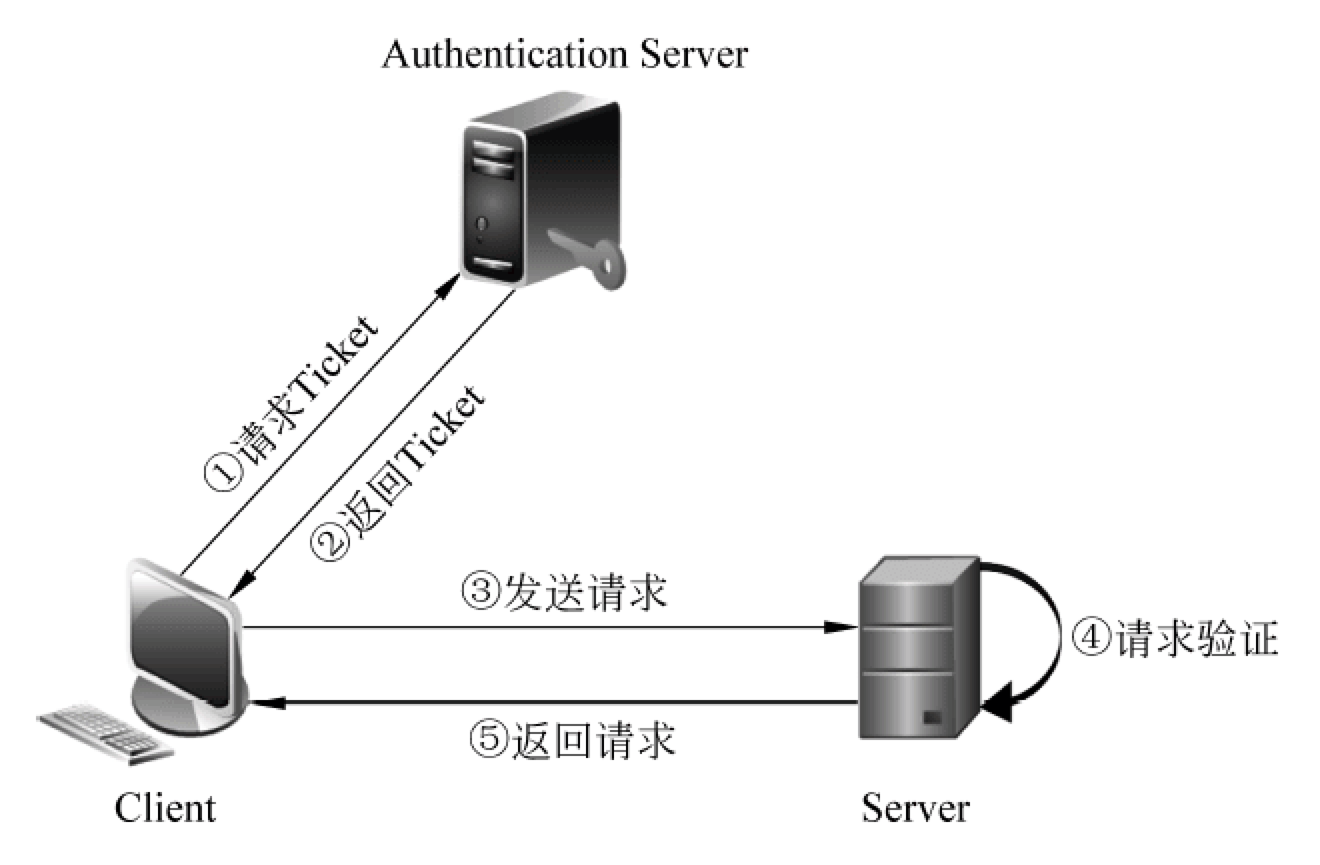


图 4.17　Kerberos 认证机制

1）请求 Ticket

Client 向认证服务器 AS 发送自己的身份信息，并请求 Ticket(包括客户标识、会话密钥、时间戳等记录信息)。

2）返回 Ticket

认证服务器 AS 在验证完 Client 身份后，会随机生成一个密码(Session Key)，并生成两个 Ticket(Ticket Granting Ticket 和 Service Ticket)返回给 Client。其中，一个 Ticket 是给 Client 的，另一个 Ticket 是给 Server 的(认证服务器 AS 并不是把 Ticket 直接给 Server，而是先交给 Client，再由 Client 交给 Server)。

3）发送请求

Client 拿到第二步中的两个 Ticket 后，用自己的密码解开 Client Ticket，并生成一个

认证因子(Authenticator)。Client 将认证因子发给 Server 的 Ticket 同时发送给 Server。

4）请求验证

Server 收到 Ticket 和 Authenticator 之后，用自己的密码解开 Server Ticket，并解开 Authenticator。Server 对这些信息进行验证，如果信息正确，则 Client 就通过了认证。

5）返回请求

Client 认证成功后，Server 选择性地给 Client 回复一条消息来完成双向认证。

Kerberos 是一种性能较高的认证机制，并且能够进行数据加密。但对于 Hadoop 中的 HDFS 而言，用户的认证比较复杂，以及 HDFS 数据传输加密等情况，采用 Kerberos 认证需要分别对其进行配置，使得 Hadoop 更难以管理并且很容易出错。因此，如果 Hadoop 要具备生产环境真正的安全能力，可以集成第三方认证系统或采用其他有创造性的方法。

4.7　HDFS 容错机制

HDFS 文件系统在设计之初时就假设系统故障(服务器、网络及存储故障等)为常态现象，即使某些节点发生故障时，整个集群的工作都不会受到影响。因此，HDFS 要通过多方面的方法来保证其容错能力，如 HDFS 数据块的多副本存储机制、NameNode 的单点失效解决机制、在 NameNode 和 DataNode 之间维持心跳检测、检测文件块的完整性、集群的负载均衡等。

4.7.1　副本策略

一般情况下，HDFS 集群是由多个机架上的机器共同组成的一个分布式集群，机架内部机器之间的网络速度通常都会高于跨机架机器之间的网络速度，而且机架之间机器的网络通信通常也会受到上层交换机间网络带宽的限制。因此，副本的存放策略至关重要，将影响到 HDFS 的可靠性和可用性。HDFS 采用了一种基于机架感知(Rack-awareness)的副本策略来改进数据的可靠性、可用性和网络带宽的利用率。在 Hadoop 1.0 生态系统和 Hadoop 2.0 生态系统中，HDFS 默认创建的副本数为 3，其副本存储策略如下。

(1) 第一个 block 副本存放在 Client 节点所在的 DataNode 中，如果 Client 节点不在集群范围内，则这个 DataNode 是随机选择的。

(2) 第二个副本存放在与第一个 DataNode 不同的机架中随机选择的 DataNode 中。

(3) 第三个副本存放在与第一个副本所在 DataNode 同一机架的另一个 DataNode 中。

如果还有更多副本，在遵循以下限制的前提下随机放在集群的 DataNode 中。

(1) 一个节点最多放置一个副本；

(2) 满足副本数少于两倍机架数的情况下，不可以在同一机架放置超过两个副本。

这种策略减少了机架之间的数据传输，提高了 HDFS 写操作的效率。由于机架的故障率远远小于节点的故障率，所以这种策略不会影响到数据的可靠性和可用性。同时，这种副本策略对一个数据块的写操作需要传输多个该数据块的副本到不同机架，从而增加

了写数据的代价。

默认情况下，HDFS 并没有启用基于机架感知的副本策略，可在配置文件 core-site.xml 中修改相应的配置项来启动该功能。本实例以 Hadoop 2.0 生态系统为例(以 Hadoop 2.6.0 stable 版本为例)，来说明 HDFS 中基于机架感知的副本策略的启用过程(请查看本书配套资料中 Demo/RackAware 文件夹中的相关内容)。

1. 配置 core-site.xml

首先要开启 HDFS 的基于机架感知的副本策略的功能，可对 core-site.xml 中的 net.topology.node.switch.mapping.impl(机架感知实现类)、net.topology.script.file.name(机架感知脚本位置)和 net.topology.script.number.args(机架感知脚本管理的主机数)进行配置，来启用该功能，其具体的配置内容如下。

```
<property>
  <name>net.topology.node.switch.mapping.impl</name>
  <value>org.apache.hadoop.net.ScriptBasedMapping</value>
  <description>The default implementation of the DNSToSwitchMapping. It
  invokes a script specified in net.topology.script.file.name to resolve
  node names. If the value for net.topology.script.file.name is not set, the
  default value of DEFAULT_RACK is returned for all node names.
  </description>
</property>

<property>
  <name>net.topology.script.file.name</name>
  <value>/opt/hadoop-2.6.0/rack.lsp</value>
</property>

<property>
  <name>net.topology.script.number.args</name>
  <value>100</value>
  <description>The max number of args that the script configured with
  net.topology.script.file.name should be run with. Each arg is an
  IP address.
  </description>
</property>
```

2. 部署脚本到 NameNode

在 core-site.xml 中，已经配置名为 rack.lsp 的脚本，其存储路径为/opt/hadoop-2.6.0/。下面就需要创建该脚本，让 Hadoop 了解 DataNode 的拓扑结构，即每个 DataNode 节点所在的 Data Center 和 Rack ID。rack.lsp 的脚本内容如下。

```
#通过 DataNode 的 IP 地址设置机架 ID(Rack ID)
#假定为 10.x.y.z 网络,
#x 用于数据中心 (Data Center)的划分
#如 10.1.y.z 和 10.2.y.z 表示不同 Data Center
#
#y 用于机架 ID 的划分
#如 10.1.1.z 和 10.1.2.z 表示同一 Data Center 中的不同 Rack ID

ipaddr=$1
segments='echo $ipaddr | cut -f 2,3 -d '.'        --output-delimiter=/'
echo /${segments}
```

3. 重启 NameNode

如果配置成功,重启 NameNode 后,可在 NameNode 启动日志中输出如下内容。

```
2015-01-28 03:40:55,715 INFO org.apache.hadoop.hdfs.StateChange: STATE* Network topology has 1 racks
and 3 datanodes
```

也可以通过如下命令,查看对应的 Data Center 和 Rack ID 的信息。

```
[hadoop@master1 hadoop-2.6.0]$ hdfs dfsadmin -printTopology
```

运行结果如图 4.18 所示。

```
[hadoop@master1 hadoop-2.6.0]$ hdfs dfsadmin -printTopology
Rack: /168/85
   192.168.85.101:50010 (Slave1.Hadoop)
   192.168.85.102:50010 (Slave2.Hadoop)
   192.168.85.103:50010 (Slave3.Hadoop)
```

图 4.18　查看机架信息

从图 4.18 可以看出,本实例的三个 DataNode 的 IP 都在同一网段。所以通过 HDFS 的机架的结果为:一个 Data Center(168),该机架中有一个 Rack,其 Rack ID 为 85。读者可根据项目实际情况,重新制定副本策略,并对 Data Center 和 Rack 进行重新规划(需要重新定义 rack.lsp 脚本)。

4.7.2　心跳检测

HDFS 将每个文件存储成一系列的数据块,所有的数据块都是同样大小(最后一个数据块除外),并通过复制数据块来解决 HDFS 的容错。其中,每个文件的数据块大小和副本数都是可以配置的(在 hdfs-site.xml 中进行配置)。在 Hadoop 生态系统中,所有的 DataNode 都需要向集群中所有的 NameNode 注册自己,DataNode 会每隔一段时间向所有的 NameNode 发送心跳和处理来自 NameNode 的命令;NameNode 负责所有数据块的

复制，它周期性地从集群中的每个 DataNode 节点接收心跳信号（用于检测 DataNode 是否正常）和块状态报告（包含该 DataNode 上所有数据块的列表）。如果 NameNode 接收到心跳信号意味着该 DataNode 节点工作正常；如果 DataNode 不能发送心跳信息，NameNode 会标记最近没有心跳的 DataNode 为宕机状态，并且不给该 DataNode 发送任何 I/O 请求。DataNode 的宕机会造成一些数据块的副本数下降并低于指定值，NameNode 会不断检测这些需要复制的数据块，并在需要的时候重新复制。

注：引发数据块重新复制的原因很多，如 DataNode 不可用、数据副本损坏、DataNode 上的磁盘错误或者复制因子增大等。

本实例以 Hadoop 2.0 生态系统为例（Hadoop 2.6.0 stable 版本），来说明 DataNode 是如何主动向 NameNode 发送心跳，以及 NameNode 是如何处理来自 DataNode 的心跳（请查看本书配套资料中 Demo/HeartBeat 文件夹中的相关内容）。

1. 启动 DataNode 节点

启动 DataNode 节点是由 org.apache.hadoop.hdfs.server.datanode.DataNode 类来实现的，其中包括两个过程：创建 DataNode 对象和启动 DataNode 节点。首先，DataNode.main()方法会调用 DataNode.secureMain()方法，并在 secureMain 方法中调用 DataNode.createDataNode()方法，createDataNode()方法又调用了 DataNode 类中的 instantiateDataNode()方法和 runDatanodeDaemon()方法。其中，instantiateDataNode()方法用于初始化 DataNode 的大部分成员变量，即创建 DataNode 对象；runDatanodeDaemon()方法用于向 NameNode 节点注册和启动 DataNode 节点的线程，即启动 DataNode 线程（具体实现细节请查看 DataNode 类中的相关方法）。

2. 向 NameNode 发送心跳及响应

启动 DataNode 线程后，就开始执行 DataNode.startDataNode()方法，该方法启动了 DataNode 流式数据交换的服务和在各个数据节点间进行 RPC 通信的服务，并经过层层调用，启动了真正与 NameNode 进行通信的 org.apache.hadoop.hfds.server.datanode.BPServiceActor 服务。BPServiceActor 类负责与 NameNode 的 RPC 对话。BPServiceActor.run()方法内部循环执行 BPServiceActor.connectToNNAndHandshake()和 BPServiceActor.offerService()方法，来向 NameNode 周期性地发送心跳和接收来自 NameNode 的响应（Response）。其中，connectToNNAndHandshake()方法的代码内容如下。

```
private void connectToNNAndHandshake() throws IOException {
    //连接到 NameNode 并获得 NameNode 代理对象
    bpNamenode= dn.connectToNN(nnAddr);
    //第一阶段获取 NamespaceInfo
    NamespaceInfo nsInfo= retrieveNamespaceInfo();
    //校验 NamespaceInfo 是否和 HA 中的其他 NameNode 信息一致
    //并建立 blockPoolManager 和 BPOfferService 的对应关系
    bpos.verifyAndSetNamespaceInfo(nsInfo);
```

```
        //第二阶段向 NameNode 注册
        register();
    }
```

BPServiceActor.offerService()方法的代码内容如下。

```
/**
 * 循环调用"发送心跳"方法,直到 shutdown
 * 调用远程 NameNode 的方法
 * /
private void offerService() throws Exception {
    LOG.info("For namenode "+ nnAddr+ " using"
        + " DELETEREPORT_INTERVAL of "+ dnConf.deleteReportInterval+ " msec "
        + " BLOCKREPORT_INTERVAL of "+ dnConf.blockReportInterval+ "msec"
        + " CACHEREPORT_INTERVAL of "+ dnConf.cacheReportInterval+ "msec"
        + " Initial delay: "+ dnConf.initialBlockReportDelay+ "msec"
        + "; heartBeatInterval= "+ dnConf.heartBeatInterval);
    while(shouldRun()){
      try {
        final long startTime= now();
        //发送心跳或调用远程 NameNode 的方法
        if(startTime- lastHeartbeat >= dnConf.heartBeatInterval){
          lastHeartbeat= startTime;
          if(!dn.areHeartbeatsDisabledForTests()){
            HeartbeatResponse resp= sendHeartBeat();
            assert resp != null;
            dn.getMetrics().addHeartbeat(now()- startTime);
            bpos.updateActorStatesFromHeartbeat(
                this, resp.getNameNodeHaState());
            state= resp.getNameNodeHaState().getState();
            if(state== HAServiceState.ACTIVE){
              handleRollingUpgradeStatus(resp);
            }
            long startProcessCommands= now();
            if(!processCommand(resp.getCommands()))
              continue;
            long endProcessCommands= now();
            if(endProcessCommands- startProcessCommands > 2000){
              LOG.info("Took "+ (endProcessCommands- startProcessCommands)
                  + "ms to process "+ resp.getCommands().length+ " commands from NN");
            }
          }
        }
```

```
        if(sendImmediateIBR ||
            (startTime - lastDeletedReport > dnConf.deleteReportInterval)) {
          reportReceivedDeletedBlocks();
          lastDeletedReport = startTime;
        }
        List<DatanodeCommand> cmds = blockReport();
        processCommand(cmds == null ? null : cmds.toArray(new DatanodeCommand[cmds.size()]));
        DatanodeCommand cmd = cacheReport();
        processCommand(new DatanodeCommand[]{ cmd });
        if(dn.blockScanner != null) {
          dn.blockScanner.addBlockPool(bpos.getBlockPoolId());
        }
        long waitTime = dnConf.heartBeatInterval -
        (Time.now() - lastHeartbeat);
        synchronized(pendingIncrementalBRperStorage) {
          if(waitTime > 0 && !sendImmediateIBR) {
            try {
              pendingIncrementalBRperStorage.wait(waitTime);
            } catch(InterruptedException ie) {
              LOG.warn("BPOfferService for " + this + " interrupted");
            }
          }
        }                                    //synchronized
      } catch(RemoteException re) {
        String reClass = re.getClassName();
        if(UnregisteredNodeException.class.getName().equals(reClass) ||
            DisallowedDatanodeException.class.getName().equals(reClass) ||
            IncorrectVersionException.class.getName().equals(reClass)) {
          LOG.warn(this + " is shutting down", re);
          shouldServiceRun = false;
          return;
        }
        LOG.warn("RemoteException in offerService", re);
        try {
          long sleepTime = Math.min(1000, dnConf.heartBeatInterval);
          Thread.sleep(sleepTime);
        } catch(InterruptedException ie) {
          Thread.currentThread().interrupt();
        }
      } catch(IOException e) {
        LOG.warn("IOException in offerService", e);
      }
```

```
        }                                          //while(shouldRun())
      }                                            //offerService
```

3. NameNode 处理心跳

NameNode 收到 DataNode 心跳包后，除了更新该 DataNode 信息之外，还要给 DateNode 发送一些指令，如更新租约(lease recovery)、复制副本(block replication)、删除数据块(block invalidation)、更新均衡器带宽(update balancer bandwith)等命令。这些功能都是由 org. apache. hadoop. hdfs. server. blockmanagement. DatanodeManager 类中的方法来实现的。其中，DatanodeManager. handleHeartbeat ( ) 方法用来处理来自 DataNode 的心跳包，其代码内容如下。

```
/** NameNode 处理来自 DataNode 的心跳 */
  public DatanodeCommand[] handleHeartbeat(DatanodeRegistration nodeReg,
      StorageReport[] reports, final String blockPoolId,
      long cacheCapacity, long cacheUsed, int xceiverCount,
      int maxTransfers, int failedVolumes
      )throws IOException {
    synchronized(heartbeatManager){
      synchronized(datanodeMap){
        DatanodeDescriptor nodeinfo=null;
        try {
          nodeinfo=getDatanode(nodeReg);
        } catch(UnregisteredNodeException e){
          return new DatanodeCommand[]{RegisterCommand.REGISTER};
        }
        //检查当前 DataNode 状态,如果宕机,表明该节点不允许连接到 NameNode 节点
        if(nodeinfo !=null && nodeinfo.isDisallowed()){
          setDatanodeDead(nodeinfo);
          throw new DisallowedDatanodeException(nodeinfo);
        }
        if(nodeinfo==null || !nodeinfo.isAlive){
          return new DatanodeCommand[]{RegisterCommand.REGISTER};
        }
        //更新 DataNode 节点的数据
        heartbeatManager.updateHeartbeat(nodeinfo, reports,
                                         cacheCapacity, cacheUsed,
                                         xceiverCount, failedVolumes);
        if(namesystem.isInSafeMode()){
          return new DatanodeCommand[0];
        }
        //更新租约
```

```
BlockInfoUnderConstruction[] blocks=nodeinfo
        .getLeaseRecoveryCommand(Integer.MAX_VALUE);
if(blocks != null){
    BlockRecoveryCommand brCommand=new BlockRecoveryCommand(
            blocks.length);
    for(BlockInfoUnderConstruction b : blocks){
        final DatanodeStorageInfo[] storages=b.getExpectedStorageLocations();
        final List<DatanodeStorageInfo> recoveryLocations=
                new ArrayList<DatanodeStorageInfo>(storages.length);
        for(int i=0; i<storages.length; i++){
            if(!storages[i].getDatanodeDescriptor().isStale(staleInterval)){
                recoveryLocations.add(storages[i]);
            }
        }
        if(recoveryLocations.size()>1){
            if(recoveryLocations.size()!=storages.length){
                LOG.info("Skipped stale nodes for recovery : "+
                        (storages.length-recoveryLocations.size()));
            }
            brCommand.add(new RecoveringBlock(
                    new ExtendedBlock(blockPoolId, b),
                    DatanodeStorageInfo.toDatanodeInfos(recoveryLocations),
                    b.getBlockRecoveryId()));
        } else {
            brCommand.add(new RecoveringBlock(
                    new ExtendedBlock(blockPoolId, b),
                    DatanodeStorageInfo.toDatanodeInfos(storages),
                    b.getBlockRecoveryId()));
        }
    }
    return new DatanodeCommand[] { brCommand };
}
final List<DatanodeCommand> cmds=new ArrayList<DatanodeCommand>();
//复制副本指令
List<BlockTargetPair> pendingList=nodeinfo.getReplicationCommand(
        maxTransfers);
if(pendingList != null){
    cmds.add(new BlockCommand(DatanodeProtocol.DNA_TRANSFER, blockPoolId,pendingList));
}
//数据块删除指令
Block[] blks=nodeinfo.getInvalidateBlocks(blockInvalidateLimit);
if(blks != null){
```

```
                cmds.add(new BlockCommand(DatanodeProtocol.DNA_INVALIDATE,
                    blockPoolId, blks));
            }
            boolean sendingCachingCommands=false;
            long nowMs=Time.monotonicNow();
            if(shouldSendCachingCommands &&
                ((nowMs-nodeinfo.getLastCachingDirectiveSentTimeMs())>=
                    timeBetweenResendingCachingDirectivesMs)){
              DatanodeCommand pendingCacheCommand=
                  getCacheCommand(nodeinfo.getPendingCached(), nodeinfo,
                    DatanodeProtocol.DNA_CACHE, blockPoolId);
              if(pendingCacheCommand !=null){
                cmds.add(pendingCacheCommand);
                sendingCachingCommands=true;
              }
              DatanodeCommand pendingUncacheCommand=
                  getCacheCommand(nodeinfo.getPendingUncached(), nodeinfo,
                    DatanodeProtocol.DNA_UNCACHE, blockPoolId);
              if(pendingUncacheCommand !=null){
                cmds.add(pendingUncacheCommand);
                sendingCachingCommands=true;
              }
              if(sendingCachingCommands){
                nodeinfo.setLastCachingDirectiveSentTimeMs(nowMs);
              }
            }

            blockManager.addKeyUpdateCommand(cmds, nodeinfo);
            //check for balancer bandwidth update
            if(nodeinfo.getBalancerBandwidth()>0){
              cmds.add(new BalancerBandwidthCommand(nodeinfo.
              getBalancerBandwidth()));
              nodeinfo.setBalancerBandwidth(0);
            }
            if(!cmds.isEmpty()){
              return cmds.toArray(new DatanodeCommand[cmds.size()]);
            }
          }
        }
        return new DatanodeCommand[0];
      }
```

4.7.3 HDFS HA

HDFS HA(High Availability)为HDFS的高可用性,是HDFS系统对外正常提供服务时间的百分比。NameNode的可用性是影响HDFS高可用性的重要因素,它负责整个HDFS文件系统的控制与管理,如果NameNode发生故障,将导致HDFS无法正常对外提供服务。因此,HDFS的HA主要由NameNode的高可用性决定的,即提高NameNode可靠性,减少NameNode故障恢复时间。在Hadoop 1.0生态系统中,HDFS存在单点故障,即每个集群只有一个NameNode,如果NameNode宕机,将导致整个集群无法使用。在Hadoop 2.0生态系统中,解决了NameNode单点故障问题,一旦主NameNode出现故障,可以迅速切换至备用NameNode,从而实现不间断对外提供服务。

注:很多读者把高可用性与高可靠性混淆,高可靠性是对系统自身而言,是系统可靠程度的一个指标;高可用性是从系统对外的服务角度而言,是系统对外正常服务能力的度量,主要强调系统中止服务后迅速恢复的能力。如果一个可靠性很高的系统,中止服务后,恢复时间很长,则该系统的可用性不高;如果一个可靠性不高的系统,在服务中止后,可迅速恢复,则该系统的可用性高。

1. HDFS HA方案及特点

为了提高HDFS的HA可从两方面考虑,即提高NameNode的可靠性和减少NameNode故障恢复时间。在NameNode的可靠性方面,可对NameNode进行备份;在NameNode故障恢复方面,可对NameNode自身的启动过程进行优化来减少启动时间,或者提供一个NameNode的热备份节点,当主NameNode节点故障时,切换为备NameNode节点,主备切换时间为NameNode故障恢复时间。目前,各大公司和Hadoop社区提出了多种改进HDFS HA的方案,常用的方案主要有以下几种。

1) NameNode多目录存储

该方案利用Hadoop自身的Failover措施来实现对元数据的备份,即通过配置文件hfds-site.xml设置配置项dfs.name.dir(Hadoop 1.0生态系统)或dfs.namenode.name.dir(Hadoop 2.0生态系统),将NameNode维护的元数据保存到多个目录,并且备份一份到远程的NFS(Network File System)目录。当NameNode发生故障时,可通过另一台NameNode读取NFS目录中的元数据进行恢复工作。该方案解决了元数据保存的可靠性问题,是Hadoop自带的机制,只需要通过简单的配置即可实现。该方案的不足之处在于写入NFS增加了系统开销,而且NameNode故障恢复时间与文件系统规模成正比,恢复过程时间较长,只能算是一种备份方案,并不是真正意义上的HA方案。

2) SecondaryNameNode

该方案启动一个SecondaryNameNode节点,该节点定期从NameNode节点上下载元数据信息(fsimage)和日志文件(edits),并进行合并更新,生成新的fsimage(该fsimage是SecondaryNameNode下载时刻的元数据的Checkpoint),从而实现对NameNode的备份。当NameNode故障时,可以通过SecondaryNameNode进行恢复。该方案解决了元数据在Checkpoint阶段元数据的可靠性问题,是Hadoop自带机制,只需要简单配置即

可实现。该方案的不足之处在于 SecondaryNameNode 保存的只是 Checkpoint 时刻的元数据，通过 Checkpoint 恢复的元数据并不是 HDFS 的最新数据，存在一致性问题。而且 NameNode 故障恢复时间与文件系统规模成正比，恢复过程时间较长，只能算是一种备份方案，并不是真正意义上的 HA 方案。

3）CheckpointNode

该方案与 SecondaryNameNode 的原理基本相同，利用了 HDFS 的 Checkpoint 机制进行备份，通过一个 CheckpointNode 节点定期从 PrimaryNameNode 节点下载元数据信息进行合并（fsimage 和 edits），形成最新的 Checkpoint，并上传到 NameNode 进行更新。该方案的不足之处在于 Checkpoint 备份没有与 NameNode 实时同步，恢复后的元数据有可能不是 NameNode 发生故障时最新的元数据信息，有可能造成数据不一致问题，而且备份节点切换时间比较长。因此，该方案也只能算是一种备份方案，并不是真正意义上的 HA 方案。

4）Backup Node

该方案利用新版本 Hadoop 自身的 Failover 措施，配置一个 Backup Node 节点，该节点在内存和磁盘中保存了 NameNode 最新的元数据。当 NameNode 发生故障时，可读取 Backup Node 节点中最新的元数据信息进行恢复。该方案解决了元数据的数据一致性问题，是 Hadoop 自带机制，只需要简单配置即可实现。该方案的不足之处在于当 NameNode 发生故障时，只能通过重启 NameNode 的方式来恢复服务，仍然需要一定的切换时间。

5）Facebook 的 AvatarNode

Facebook 的 AvatarNode 存在两个 NameNode 节点，分别是 Active Node 和 Standby Node。其中，Active Node 用于对外提供服务，Standby Node 处于安全模式，在内存中保存 Active Node 的最新元数据信息。Active Node 和 Standby Node 通过共享的 NFS 进行交互。当 Active Node 故障时，管理员可通过一条命令将 Standby Node 转换为 Active Node，大大减少了 NameNode 的故障恢复时间，而且保证了元数据信息的一致性。该方案提供了一种热备份，使得切换时间大大缩短，是一种真正的 HA 方案。该方案的不足之处在于 Standby Node 与 Active Node 之间并不是自动切换，当 NameNode 故障时，需要管理员确认后，由管理员进行手动切换。

6）社区开发的 HDFS HA

社区开发的 HDFS HA 方案与 Facebook 的 AvatarNode 原理基本相同，区别在于对 Active Node 和 Standby Node 的共享日志的处理不同。早期的社区 HFDS HA 版本是将 Active Node 和 Standby Node 共享 NFS，该方案需要专用存储设备，在使用上有一定限制。随后，采用了基于 BookKeeper 的日志存储方案，即通过 ZooKeeper 的 BookKeeper 实现日志的高可靠共享存储，该方案对 ZooKeeper 依赖较大，配置比较复杂。目前，最新 Hadoop 版本采用了基于 QJM（Qurom Journal Manager）的共享日志方案，即通过 $2N+1$ 个 Journal Node 节点存储日志，该方案独立性好，配置比较简单，是社区主推的方案。

在实际应用中，往往采用两种以上的组合方式来保证 HDFS 的 HA，如 NameNode

多目录存储与 SecondaryNameNode（该方案适用于目前 Hadoop 所有版本）、NameNode 多目录存储与 Backup Node（该方案适用于 Hadoop 0.21.0 版本以上）、NameNode 多目录存储与 AvatarNode（该方案适用于 Hadoop 的特定版本）。表 4.5 给出了这几种常用 HDFS 的 HA 方案的比较。

表 4.5　常用 HDFS 的 HA 方案的比较

方　案	恢复时间	元数据一致性	使用复杂度
NameNode 多目录存储	长	一致	低
SecondaryNameNode	长	不一定	中
CheckpointNode	长	不一定	中
BackupNode	中	一致	中
Facebook AvatarNode	短	一致	高
社区 HDFS HA	短	一致	高

2. 基于 QJM 的 HDFS HA 实现机制

在 Hadoop 2.0 生态系统中，Hadoop 开源社区引入了 QJM（Quorum Journal Manager）和 NFS 用于 HDFS 的 HA 机制的实现，其实现原理及架构与 Facebook 的 AvatarNode 非常相似，都包括两个 NameNode（Active NameNode 和 Standby NameNode），一个共享存储，若干个 DataNode 和 Client。其中，Active NameNode 是对外提供服务的 NameNode；Standby NameNode 通过共享存储与 Active NameNode 进行元数据同步的 NameNode。当 Active NameNode 失效时，可将 Standby NameNode 切换为 Active 状态，接替失效的 NameNode 对外服务。它们的不同之处在于日志的共享方案不同。基于 QJM 的日志共享的基本原理（Paxos 算法）是用 $2N+1$ 台 Journal Node 存储日志，每次写数据操作有大多数（$W \geqslant N+1$）返回成功时，就认为该次写入日志成功，实现机制如图 4.19 所示。

在图 4.19 中，DN 表示 DataNode；NN 表示 NameNode；JN 表示 Journal Node；ZK 表示 ZooKeeper。Active NameNode 用于接收来自 Client 的 RPC 请求并处理，并写自己的 editlog 和共享存储上的 editlog，接收 DataNode 的块报告（Block Reports）、更新块位置（Block Location Updates）和心跳（Heartbeat）。Standby NameNode 同样用于接收来自 DataNode 的相关信息，同时会从共享存储的 editlog 上读取并执行这些 log 操作，使得自己的 NameNode 中的元数据信息和 Active NameNode 中的元数据信息是同步的。Journal Node 用于 Active NameNode 和 Standby NameNode 之间的通信，而且需要保证一个集群中至少运行三个 Journal Node 守护进程，从而使得系统有一定的容错能力。如果集群中要设置 N 个 Journal Node 节点（最好设置成奇数个 Journal Node），系统最多能容忍（$N-1$）/2 个 Journal Node 节点崩溃。在手动切换的基础上，社区又开发了基于 ZooKeeper 的 ZKFC（ZooKeeper Failover Controller）自动切换解决方案，每个 Active Node 和 Standby Node 各有一个 ZKFC 进程监控 NameNode 的健康状况，当 Active

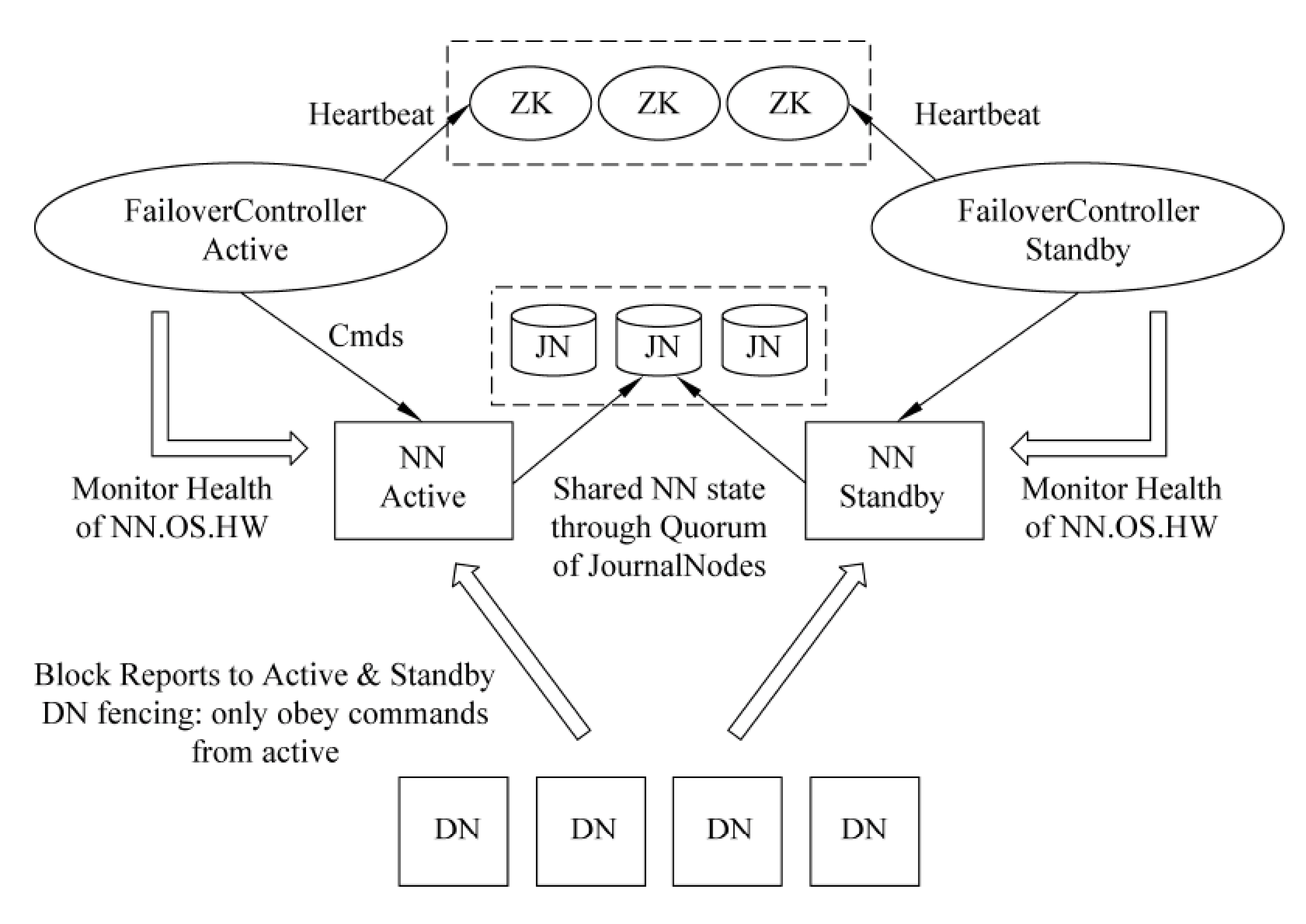


图 4.19 基于 QJM 的 HDFS HA 实现机制

（图片来源：HDFS-1623 设计文档）

Node 出现问题时，自动将 Standby Node 切换为 Active Node。

3. 配置基于 QJM 的 HDFS 的 HA

在 Hadoop 2.0 生态系统中，Hadoop 社区提供了两种 HDFS HA 的实现方式，分别是基于 QJM 的 HDFS HA 和基于 NFS 的 HDFS HA。本实例将以基于 QJM 的 HDFS HA 的配置为例（Hadoop 2.6.0 stable 版本），介绍如何实现 HDFS HA（请查看本书配套资料中 Demo/HDFS HA 文件夹中的相关内容）。

1）确定集群结构

HDFS HA 的配置向后兼容，允许已存在的单 NameNode 配置在没有任何改动的情况下工作，新加入的 NameNode 节点和集群中所有节点拥有着相同的配置，不必为不同的机器设置不同的配置文件。因此，在表 3.7 集群规划的基础上只需要增加一个 NameNode 节点即可。集群中节点角色规划如表 4.6 所示。

表 4.6 集群节点角色分配

机器名称	角色	IP 地址	硬件参数	操作系统
Master1.Hadoop	Active NameNode JournalNode	192.168.85.100	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave1.Hadoop	DataNode JournalNode	192.168.85.101	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave2.Hadoop	DataNode	192.168.85.102	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7

续表

机器名称	角　色	IP 地址	硬件参数	操作系统
Slave3. Hadoop	DataNode	192. 168. 85. 103	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Master2. Hadoop	Standby NameNode JournalNode	192. 168. 85. 99	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7

在一个 HA 集群中备份的 NameNode 也要坚持 Namespace 的状态,那么就没有必要去运行一个 SecondaryNameNode、CheckpointNode 或者 BackupNode 在集群当中。对于 NameNode 和 SecondaryNameNode 相互独立的两个节点,为了方便非 HA 的集群可以实现 HA,可以把以前的 SecondaryNameNode 节点作为 Standby NameNode,并且实现硬件的重用。在之前表 3. 6 的集群规划中,将 NameNode 和 SecondaryNameNode 两个角色都放在 Master1. Hadoop 节点上。因此,本实例将增加一个 Master2. Hadoop 节点作为 Standby NameNode 角色节点。在 HA 集群中,JournalNode 用于 Active NameNode 和 Standby NameNode 之间的通信,并且至少需要三个以上节点,所以将 Master1. Hadoop、Master2. Hadoop 和 Slave1. Hadoop 节点作为 JournalNode 角色节点。

2) 配置 hdfs-site. xml 文件

要实现 HDFS 的 HA,必须添加相应的配置项到配置文件 hdfs-site. xml 中,并将配置文件同步到所有的节点上(可以使用 rsync 将配置文件同步到节点上),具体的配置内容如下。

```
<configuration>
<!--"dfs.nameservices"定义 nameservices 的名字,这个名字可以任意定义,这里定义成
"cluster1"。-->
        <property>
                <name>dfs.nameservices</name>
                <value>cluster1</value>
        </property>
<!--"dfs.ha.namenodes.[nameservice ID]"标识每个 NameNode,每个 NameNode 的标识必须唯一,并且最
多有两个 NameNode。这里将两个 NameNode 分别取名为 nn1,nn2。-->
        <property>
                <name>dfs.ha.namenodes.cluster1</name>
                <value>nn1,nn2</value>
        </property>
<!--"dfs.namenode.rpc-address.[nameservice ID].[name node ID]" 定义每个 NameNode 的 IP 或
Hostname 以及端口号,这里定义 nn1 和 nn2 的 RPC 通信地址 -->
        <property>
                <name>dfs.namenode.rpc-address.cluster1.nn1</name>
                <value>Master1.Hadoop:9000</value>
        </property>
```

```
        <property>
                <name>dfs.namenode.rpc-address.cluster1.nn2</name>
                <value>Master2.Hadoop:9000</value>
        </property>
<!--"dfs.namenode.http-address.[nameservice ID].[name node ID]"定义 HTTP 服务的端口号,这里定义 nn1 和 nn2 的 http 通信地址 -->
        <property>
                <name>dfs.namenode.http-address.cluster1.nn1</name>
                <value>Master1.Hadoop:50070</value>
        </property>
        <property>
                <name>dfs.namenode.http-address.cluster1.nn2</name>
                <value>Master2.Hadoop:50070</value>
        </property>
<!--"dfs.namenode.shared.edits.dir"共享存储目录的位置。这是配置备份节点需要随时保持同步活动节点所做更改的远程共享目录,只能配置一个目录,这个目录挂载到两个 NameNode 上都必须是可读写的,且必须是绝对路径。-->
        <property>
                <name>dfs.namenode.shared.edits.dir</name>
                <value>qjournal://Master1.Hadoop:8485;Master2.Hadoop:8485;Slave1.Hadoop:8485/
                cluster1</value>
        </property>
<!--"dfs.client.failover.proxy.provider.[nameservice ID]" HDFS Client 用来和活动的 NameNode 进行联系的 Java 类。配置的 Java 类是用来给 HDFS Client 判断哪个 NameNode 节点是活动的,当前是哪个 NameNode 处理 Client 端的请求。目前 Hadoop 唯一的实现类是 ConfiguredFailoverProxyProvider。
-->
        <property>
                <name>dfs.client.failover.proxy.provider.gagcluster</name>
                <value> org. apache. hadoop. hdfs. server. namenode. ha.
                ConfiguredFailoverProxyProvider</value>
        </property>
<!--"dfs.ha.fencing.methods"用于停止活动 NameNode 节点的故障转移期间的脚本或 Java 类的列表。在任何时候只有一个 NameNode 处于活动状态,对于 HA 集群的操作是至关重要的。Hadoop 提供了 shell 和 sshfence 两种方法,要实现自己的方法,请看 org.apache.hadoop.ha.NodeFencer 类。-->

        <property>
                <name>dfs.ha.fencing.methods</name>
                <value>sshfence</value>
        </property>
<!—"dfs.ha.fencing.ssh.private-key-files"用来设置隔离机制时需要 ssh 免密码登录 -->
        <property>
                <name>dfs.ha.fencing.ssh.private-key-files</name>
```

```
                <value>/home/hadoop/.ssh/id_rsa</value>
        </property>
<!--"dfs.journalnode.edits.dir" 用来定义 NameNode 的元数据在 JournalNode 上的存放位置 -->
        <property>
                <name>dfs.journalnode.edits.dir</name>
                <value>/opt/hadoop-2.6.0/tmp/journal</value>
        </property>
<!—"dfs.namenode.name.dir"用来定义 NameNode 名称空间的存储地址-->
        <property>
                <name>dfs.namenode.name.dir</name>
                <value>/opt/hadoop-2.6.0/dfs/name</value>
        </property>
<!—"dfs.datanode.data.dir"用来定义 DataNode 数据存储地址-->
        <property>
                <name>dfs.datanode.data.dir</name>
                <value>/opt/hadoop-2.6.0/dfs/data</value>
        </property>
<!—"dfs.replication"指定数据冗余份数-->
        <property>
                <name>dfs.replication</name>
                <value>3</value>
        </property>
<!—"dfs.webhdfs.enabled"指定可以通过 Web 访问 hdfs 目录-->
        <property>
                <name>dfs.webhdfs.enabled</name>
                <value>true</value>
        </property>
</configuration>
```

3）启动 JournalNode

配置完 hdfs-site. xml 之后，需要启动 JournalNode 角色，即启动 Master1. Hadoop、Master2. Hadoop 和 Slave1. Hadoop 上的 JournalNode。在 Master1. Hadoop 的终端的具体命名如下。

```
$hadoop-daemons.sh --hostnames 'Master1.Hadoop Master2.Hadoop Slave1.Hadoop' start journalnode
```

4）NameNode 格式化

启动 JournalNode 之后，首先需要对 Master1. Hadoop 的 NameNode 进行格式化，并在格式化完成后，重启动 Master1. Hadoop 的 NameNode，即为 Active NameNode，具体命令如下。

```
$ hadoop namenode - format
15/01/31 03:01:59 INFO util.ExitUtil: Exiting with status 0
15/01/31 03:01:59 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master1.hadoop/192.168.85.100
************************************************************/
$ hadoop- daemon.sh start namenode
```

在返回的结果中，如果显示“Exiting with status 0”表示格式化成功，否则表示格式化失败（请查看 NameNode 的日志文件 logs/hadoop-hadoop-namenode-master1. hadoop. log）。

5）元数据同步

在 Master2. Hadoop 节点上对 NameNode 进行元数据同步，并启动 NameNode，即 Standby NameNode，具体命令如下。

```
$ hdfs namenode - bootstrapStandby
15/01/31 03:03:22 INFO util.ExitUtil: Exiting with status 0
15/01/31 03:03:22 INFO namenode.NameNode: SHUTDOWN_MSG:
/************************************************************
SHUTDOWN_MSG: Shutting down NameNode at master2.hadoop/192.168.85.99
************************************************************/
$ hadoop- daemon.sh start namenode
starting namenode, logging to /opt/hadoop- 2.6.0/logs/hadoop- hadoop- namenode- master2.hadoop.out
```

6）查看或切换 HA 节点

启动 Slave1. Hadoop、Slave2. Hadoop 和 Slave3. Hadoop 上的 DataNode，就可以通过 hdfs haadmin 命令来查看 HA 的状态，或者 Active NameNode 和 Secondary NameNode 之间切换，具体命令如下。

```
$ haoop- daemon.sh start datanode
//激活 Master1.Hadoop 节点上的 NameNode
$ hdfs haadmin - transitionToActive nn1

//查看 Master1.Hadoop 节点上 NameNode 状态
$ hdfs haadmin - getServiceState nn1
active
//查看 Master2.Hadoop 节点上 NameNode 状态
$ hdfs haadmin - getServiceState nn2
standby

//手动对 Master1.Hadoop 节点和 Master2.Hadoop 节点上的 NameNode 切换
```

```
$ hdfs haadmin -failover nn1 nn2
Failover from nn1 to nn2 successful

//查看 Master1.Hadoop 节点上 NameNode 状态
$ hdfs haadmin -getServiceState nn1
standby

//查看 Master2.Hadoop 节点上 NameNode 状态
$ hdfs haadmin -getServiceState nn2
active
```

通过上面的配置与测试，就成功实现了基于 QJM 的 HDFS HA。目前，本实例是通过手动切换 Active NameNode 和 Standby NameNode 的。如果要实现 Active NameNode 和 Standby NameNode 之间自动切换可使用基于 ZooKeeper 的 ZKFC（ZooKeeper Failover Controller）自动切换解决方案（需要安装 ZooKeeper，配置 hdfs-site.xml 和 core-site.xml 文件）。

注：基于 QJM 的 HDFS HA 方案采用了 Quorum Commit Protocol，并引入两个角色：QuorumJournalManager 和 JournalNode，QuourumJournalManager 通过 RPC 将 edits 日志写入 N（N 为奇数）个 JournalNode，只要有大多数（大于 $N/2$ 个）JournalNode 成功写入则任务日志写入成功。想深入了解该实现机制的读者，请查看 Hadoop 源码目录中的 org.apache.hadoop.hdfs.qjournal.* 包中的源码。

4.7.4 HDFS Federation

在 Hadoop 1.0 生态系统中，整个 HDFS 集群中只有一个命名空间（Namespace），并且只有一个 NameNode 来负责对命名空间的管理，存在单节点故障。在 Hadoop 2.0 生态系统中，HDFS 引入了 HDFS Federation，使得 HDFS 支持多个 Namespace，并且允许在 HDFS 中同时存在多个 NameNode。

1. HDFS Federation 架构

HDFS Federation 是为了解决 HDFS 单点故障而提出的 NameNode 的水平扩展方案，该方案允许 HDFS 创建多个 Namespace 以提高集群的扩展性和隔离性。HDFS Federation 架构如图 4.20 所示。

从图 4.20 可以看出，HDFS Federation 架构分为两层：Block Storage 和 Namespace。其中，Block Storage 层主要负责数据块 Block 的存储；Namespace 层主要负责命名空间的管理。HDFS Federation 使用了多个独立的 NameNode/Namespace 来使得 HDFS 的命名服务能够水平扩展，NameNode 之间相互独立且不需要相互协调。这些 NameNode 共用集群中 DataNode 上的存储资源，并且每个 NameNode 的 Namespace 被创建时都会定义一个存储池 BlockPoolID（跨集群的全局唯一）。每个 NameNode 都可以单独对外提供服务。DataNode 每隔一段时间会按照存储池 ID 向其对应的 NameNode 发送心跳信息，

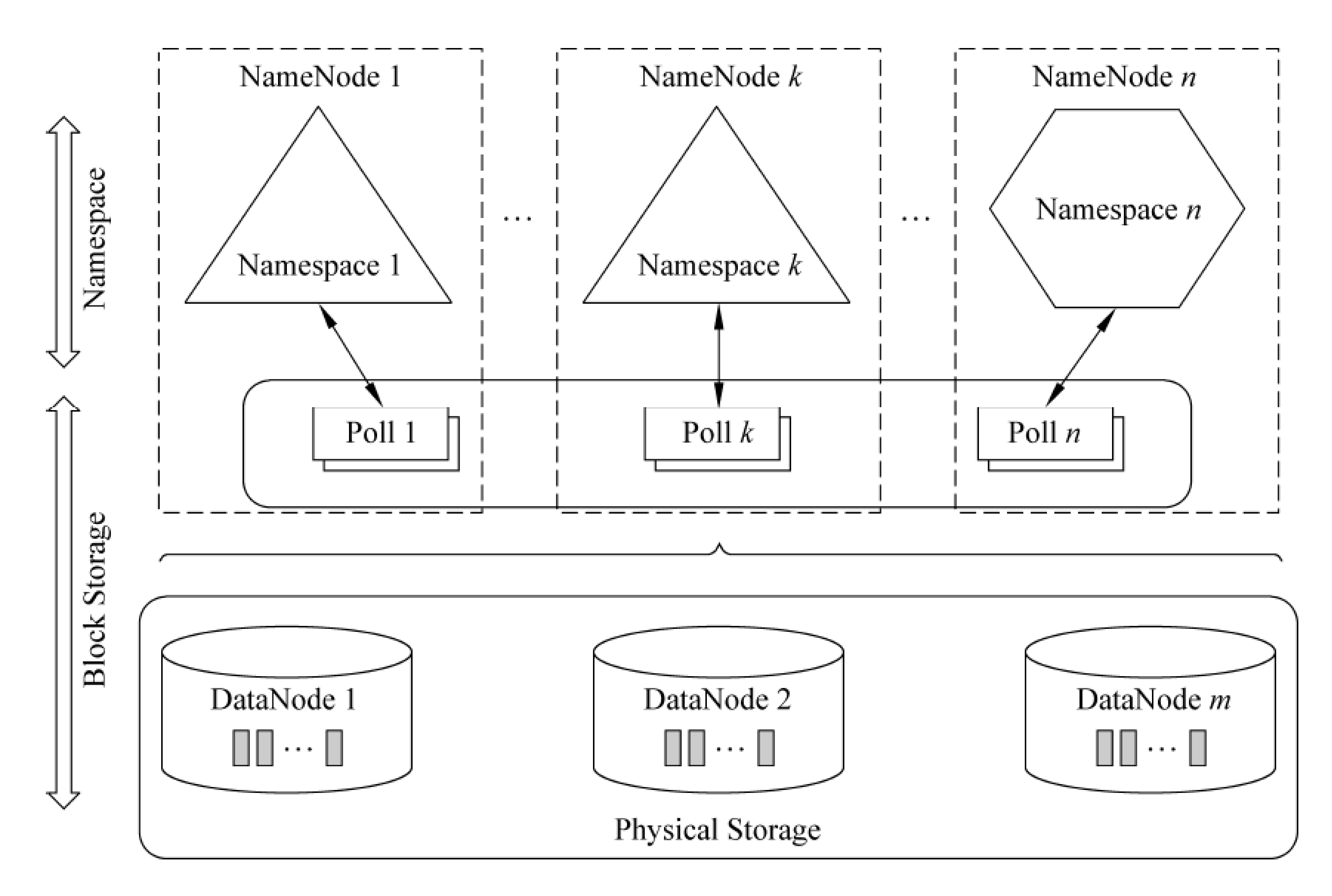


图 4.20　HDFS Federation 架构

（图片来源：HDFS-1052 设计文档）

同时向所有 NameNode 发送块状态报告，并处理来自 NameNode 的命令。HDFS Federation 存在多个 Namespace，划分和管理这些 Namespace 采用了 Client Side Mount Table，即将各个 Namespace 挂载到全局 mount-table 中，就可以做到将数据全局共享；同样的 Namespace 挂载到个人的 mount-table 中，这就成为应用程序可见的命名空间视图。

HDFS Federation 具有良好的向后兼容性，对已有的单 NameNode 的部署配置不需要任何改变就可以继续工作，并且提供了统一的块存储管理和良好的扩展性，保证了资源利用率。但是 HDFS Federation 并没有完全解决单点故障问题，虽然 NameNode/Namespace 存在多个，但是从单个 NameNode/Namespace 来看，仍然存在单点故障，而且 HDFS Federation 采用了 Client Side Mount Table 进行划分和管理 Namespace，在负载均衡方面需要人工介入。

2. HDFS Federation 配置

由于 Hadoop 1.0 生态系统并不支持 HDFS Federation。因此，本实例以在 Hadoop 2.0 生态系统为例（Hadoop 2.6.0 stable 版本），介绍如何通过配置 HDFS，实现 HDFS Federation。在实际使用过程中，HDFS HA 和 HDFS Federation 是一起配合使用的（如图 4.7 所示的 HDFS 架构）。为了方便读者理解，本实例单独使用 HDFS Federation 进行单独配置介绍（请查看本书配套资料中 Demo/Federation 文件夹中的相关内容）。

1）确定集群架构

假设本实例没有使用 HA，而是直接使用 Master1.Hadoop 和 Master2.Hadoop 组成了 Federation 集群，对应的 Namespace 的逻辑名称分别为 ns1 和 ns2，并在 Client 端进行了挂载表的映射 Client Side Mount Table，把/share 映射到 ns1，把/user 映射到 ns2。本

实例的 Federation 集群架构如图 4.21 所示。

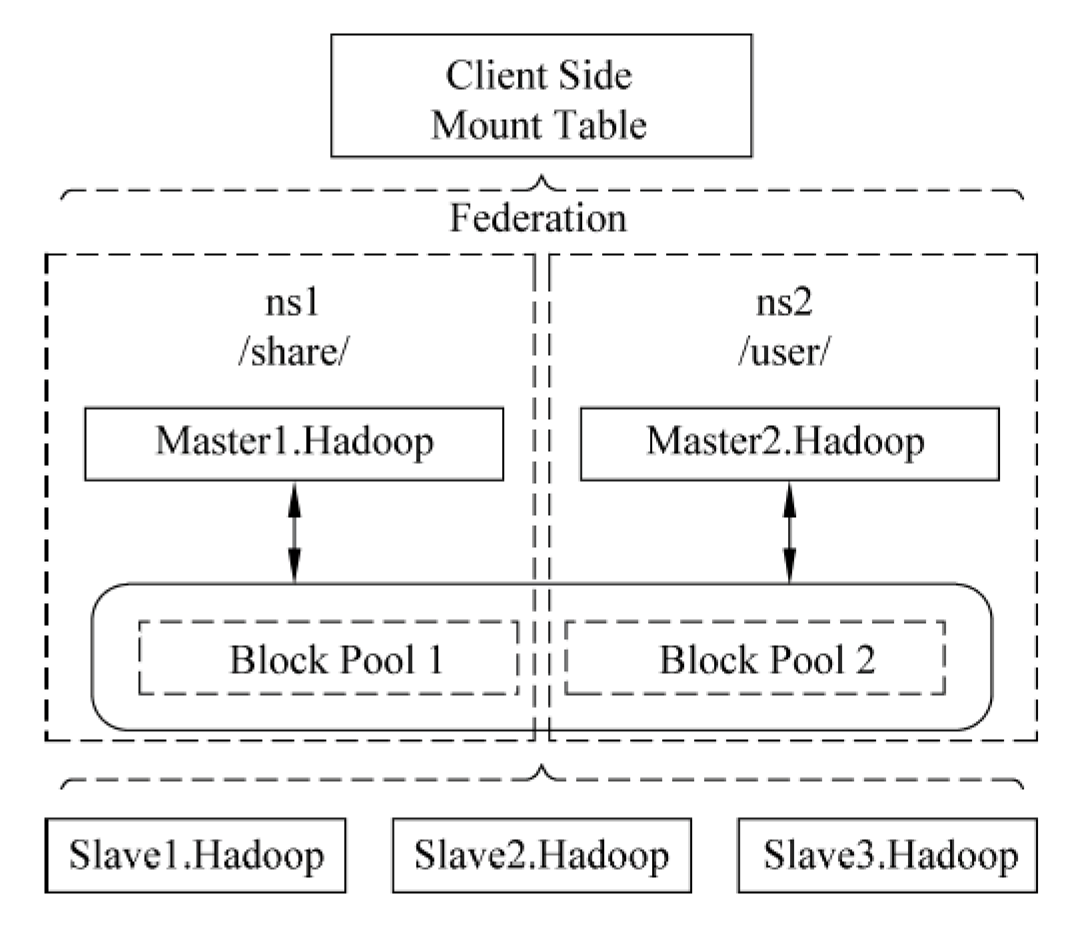


图 4.21 Federation 集群架构

2）配置 core-site.xml

HDFS Federation 使用 Client Side Mount Table 实现了为各个 Namespace 提供一个统一的视图(viewfs)。该 viewfs 实际上提供了一种映射关系，将一个全局目录映射到某个具体的 NameNode/Namespace 目录上，在 core-site.xml 中需要修改的配置如下。

```
<configuration xmlns:xi="http://www.w3.org/2001/XInclude">
  <xi:include href="mountTable.xml"/>

<!--"fs.defaultFS"用来定义整个 Federation 集群对外提供服务的 Namespace 逻辑名称，这里不再
使用 hdfs，而是新引入的 viewfs-->
    <property>
      <name>fs.defaultFS</name>
      <value>viewfs://nsX</value>
    </property>
</configuration>
```

3）配置 mountTable.xml

在 core-site.xml 文件中，引入了一个配置文件 mountTable.xml。该文件用于配置 Client Mount Table，即将虚拟路径映射到某个 Namespace 及其物理子目录。本实例将/share 映射到 ns1 的/real_share，/user 映射到 ns2 的/real_user，其配置内容如下。

```
<configuration>
    <property>
        <name>fs.viewfs.mounttable.nsX.link./share</name>
        <value>hdfs://ns1/real_share</value>
    </property>
    <property>
```

```
        <name>fs.viewfs.mounttable.nsX.link./user</name>
        <value>hdfs://ns2/real_user</value>
    </property>
</configuration>
```

4）配置 hdfs-site. xml

hdfs-site. xml 中需要给出每个 Namespace 的具体信息，需要添加的配置内容如下。

```
<!--"dfs.nameservices"提供服务的 Namespace 逻辑名称，与 core-site.xml 或 mountTable.xml 里的对
应-->
<property>
    <name>dfs.nameservices</name>
    <value>ns1,ns2</value>
    <description></description>
</property>

<property>
    <name>dfs.namenode.rpc-address.ns1</name>
    <value>Master1.Hadoop:9000</value>
</property>

<property>
    <name>dfs.namenode.http-address.ns1</name>
    <value>Master1.Hadoop:50070</value>
</property>

<property>
    <name>dfs.namenode.rpc-address.ns2</name>
    <value>Master2.Hadoop:9000</value>
</property>

<property>
    <name>dfs.namenode.http-address.ns2</name>
    <value>Master2.Hadoop:50070</value>
</property>
```

最后，在 NameNode 上建立好 Client 端挂载表映射的目标物理路径，如/real_share 和/real_user，并格式化 NameNode 之后，集群中就有两个 NameNode：Master1. Hadoop 和 Master2. Hadoop。其中，Master1. Hadoop 的 Namespace 为 ns1，管理/share 目录；Master2. Hadoop 的 Namespace 为 ns2，管理/user 目录。可使用 hdfs 命令访问 HDFS 上的文件，结果如下所示。

```
$hdfs dfs -ls /
Found 2 items
-r-xr-xr-x   -hadoop hadoop          0 2015-02-02 02:55 /share
-r-xr-xr-x   -hadoop hadoop          0 2015-02-02 02:55 /user
```

通过本实例，并结合 HDFS HA 实例，其实 HA 和 Federation 配置的关键就在于 hdfs-site.xml 中相应配置项的组合，读者可自行实践 HDFS HA 和 Federation。如果还需要更深入了解实现细节的话，可以详细阅读设计文档或代码。

4.8 HDFS 快照机制

在 Hadoop 2.0 生态系统中（Hadoop 2.1.0 Beta 版本以上），HDFS 提供了快照（SnapShot）功能，可对 HDFS 目录创建快照，创建之后不管后续目录发生什么变化，都可以通过 SnapShot 恢复到某一时刻的文件和目录结构。

4.8.1 快照原理

一个快照（SnapShot）是一个全部文件系统或某个目录在某一时刻的镜像，用于数据备份、回滚，从而防止因用户的误操作导致集群出现问题。在 HDFS 中，快照功能是通过对 NameNode 节点的操作来实现的，即在每个目标节点下面创建一个 SnapShot 节点，后续任何子节点的变化都会同步记录到该 SnapShot 节点上。例如，当 HDFS 启用 SnapShot 功能后，文件目录树结构如图 4.22 所示。

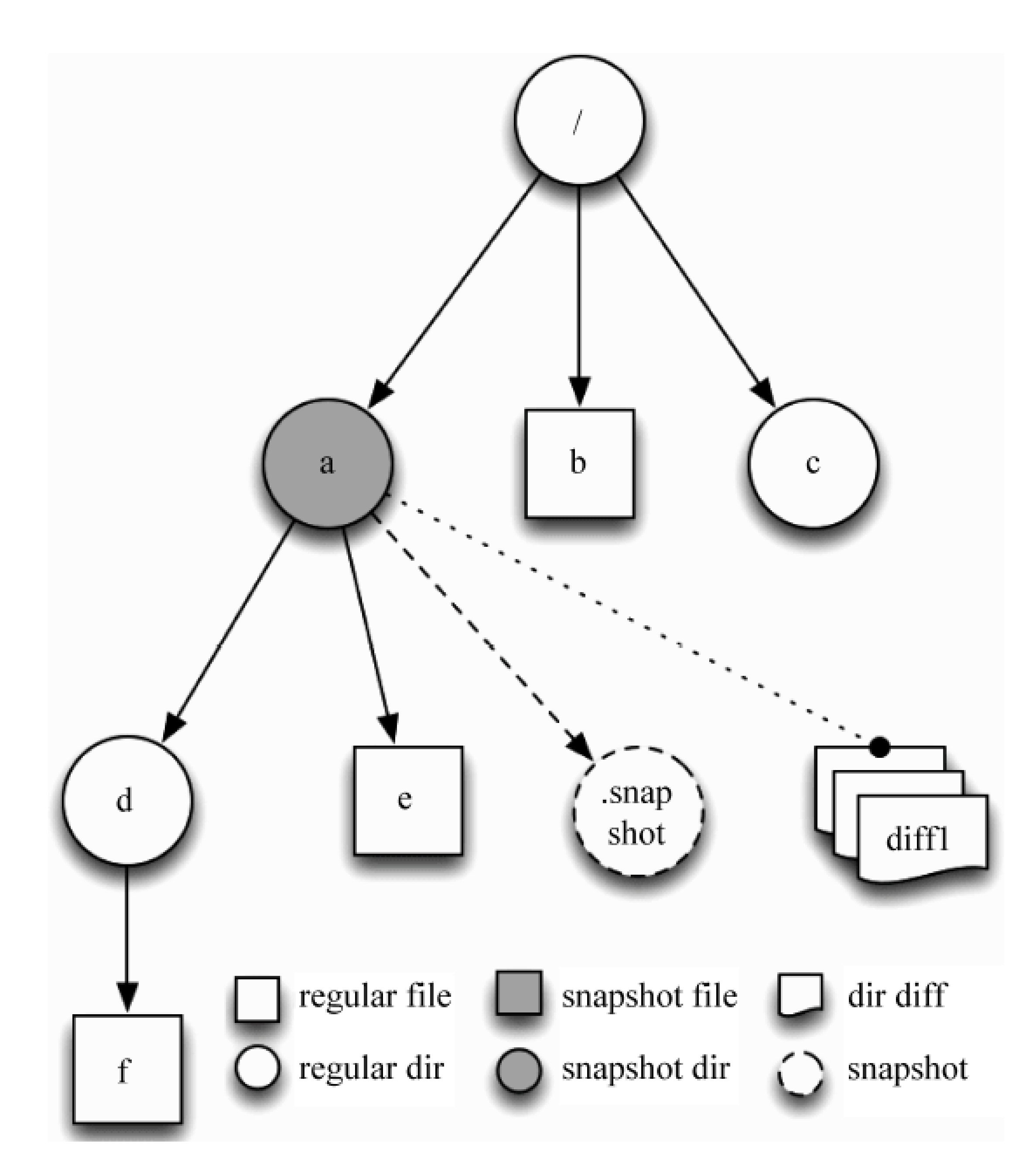


图 4.22 HDFS 开启 SnapShot 后的文件目录结构

图 4.22 表示启用了 a 的 SnapShot 功能。其中，.snapshot 是一个虚拟节点，保存了所有 SnapShot 列表；diff 保存了当前节点的变化，一个 SnapShot 对应一个 diff。当创建一个 SnapShot 时，实际的创建动作仅仅是在目录对应的节点上添加个快照标签，而数据复制的动作将被延迟到实际修改时，如图 4.23 所示。

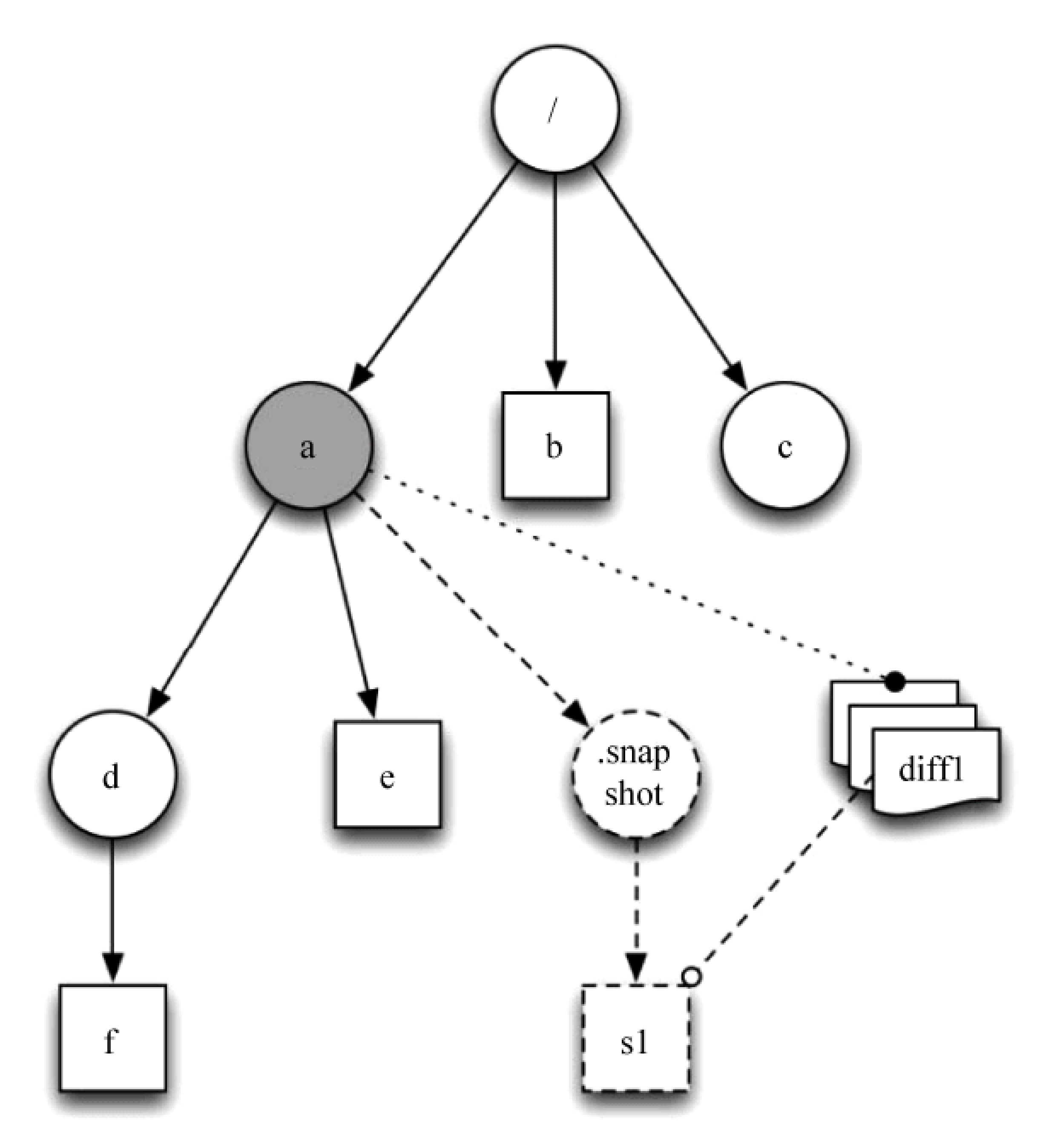


图 4.23　在可快照目录 a 上创建快照 s1 的过程

从图 4.23 可以看出，SnapShot 的创建是瞬间完成的，除节点遍历耗时之后，其快照创建操作的时间复杂度为 $O(1)$，而且在 a 目录上初始创建快照时并不占用额外内存，只有在后续对 a 目录中的内容进行修改时，才会占用额外的 NameNode 内存空间，其内存占用为 $O(M)$，M 为修改的数量。当在 a 目录上进行添加、删除文件或目录时，如果是直接子节点，都会将节点转换为 SnapShot 版本；如果是对子孙节点进行操作，首先将子孙节点转变为 SnapShot 版本，然后将父节点变为 SnapShot 版本，同时将子孙节点版本加入到直接父节点的 diff 列表中。为了能够通过同一个 SnapShot 找到当时的文件，需要将新的 diff 指向到老的 SnapShot 版本上。如图 4.24 所示为在快照目录 a 上添加、删除文件的过程。

从图 4.24 可以看出，在 a 目录中直接添加 g 节点和直接删除 e 节点，只需要将节点转换为 SnapShot 版本，其 diff 会记录添加和删除节点信息；在 a 目录中删除子孙节点 f 时，先将子孙节点 f 转换为 SnapShot 版本(diff2)，同时把 diff2 添加到直接父节点的 diff 列表中。当需要读取快照文件时，HDFS 通过反向执行所记录的修改操作，即 SnapShot 数据由当前数据减去已经修改的部分：SnapShot Data＝current data－modification，然后进行读取操作。基于这种方式实现的快照，并不会影响正常的 HDFS 操作。因为当前 HDFS 文件的读取流程并没有改变，而额外的维护操作也被限制在快照文件中。

4.8.2　适用场景

SnapShot 使用一种可选的 Copy-On-Write 方式创建文件的“逻辑”副本，因此，快照创建的速度很快，资源开销也较小，适合用于作为常规备份手段，以防止由于误删除引起

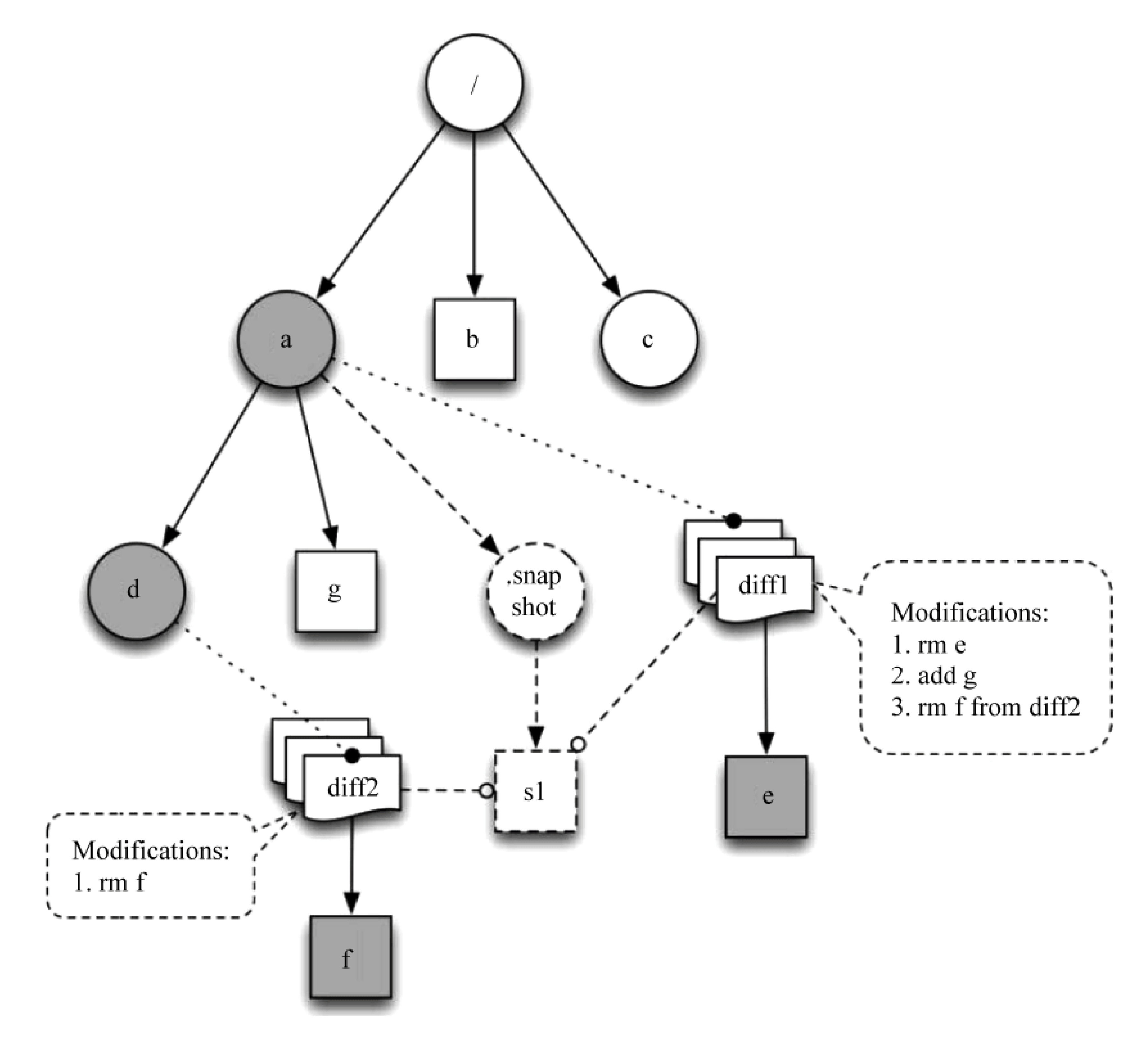


图 4.24　在快照目录 a 上添加、删除文件过程

的数据丢失。SnapShot 在下面的场景中比较适用。

1. 防止用户误操作

可以通过滚动的方式周期性地设置一个只读快照，这样就可以在 HDFS 上有若干份只读快照。即使用户意外操作删除或修改了一个文件，也可以使用包含该文件的最新只读快照来进行恢复。

2. 测试/试验

当用户想在某数据集上进行测试应用程序，但又担心测试应用程序会覆盖或损坏原来的数据集时，可以为用户设置一个该数据集的快照供应用测试使用，在快照上的数据操作不会影响到原有数据集。

3. 常规按时备份

当用户需要按固定时间间隔(例如每天)的方式进行数据备份时，可通过按照定时任务创建文件快照，实现常规的按时备份数据，同时也提升了后续外部备份时的数据一致性(快照操作耗时要远低于外部备份)。

4.8.3　基本操作

当要使用 HDFS 的快照功能时，首先要启动快照功能，其命令格式为：

```
hdfs dfsadmin -allowSnapshot <path>
```

表示可对某一个路径（可以是根目录/，某一目录或者文件）开启快照功能。例如，对根目录开启快照功能的命令如下。

```
$ hdfs dfsadmin -allowSnapshot /                    //开启根目录"/"快照功能
Allowing snaphot on / succeeded                     //表示根目录快照功能开启成功
```

开启快照功能之后，就可以创建一个 SnapShot，其命令格式为：

```
hdfs dfs -createSnapshot <path> [<snapshotName>]
```

其中，<path>为 SnapShot 路径；<snapshotName>为快照名，默认快照名的格式为's'yyyyMMdd-HHmmss.SSS，如 s20150515-084657.639。如在根目录创建一个名为 s1 的快照，其命令如下。

```
$ hdfs dfs -createSnapshot / s1                     //在根目录下创建一个名为 s1 的快照
Created snapshot /.snapshot/s1                      //表示快照创建成功
```

为了方便演示快照的一些其他常用命令，本实例在创建快照 s1 的基础上创建两个文件：f1 和 f2，并新建快照 s2，其命令如下。

```
$ hdfs dfs -touchz /f{1,2}                          //创建 f1 和 f2 文件
$ hdfs dfs -createSnapshot / s2                     //新建快照 s2
Created snapshot /.snapshot/s2                      //快照 s2 创建成功
```

删除文件 f2，并重新创建一个快照 s3，查看快照 s2 和快照 s3 的内容，其具体命令如下。

```
##查看快照 s2 的内容
$ hdfs dfs -ls -R /
-rw-r--r--   3 hadoop supergroup          0 2015-01-27 01:37 /f1
-rw-r--r--   3 hadoop supergroup          0 2015-01-27 01:37 /f2
drwxr-xr-x   - hadoop supergroup          0 2015-01-06 21:24 /input
-rw-r--r--   3 hadoop supergroup  195257604 2015-01-06 21:24 /input/hadoop-
2.6.0.tar.gz
-rw-r--r--   3 hadoop supergroup         21 2015-01-05 01:47 /input/wordcount
drwxr-xr-x   - hadoop supergroup          0 2015-01-05 01:51 /output
-rw-r--r--   3 hadoop supergroup          0 2015-01-05 01:51 /output/_SUCCESS
```

```
-rw-r--r--   3 hadoop supergroup            27 2015-01-05 01:51 /output/part-00000

##删除文件 f2
$hdfs dfs -rm /f2                         //删除根目录下 fs 文件
15/01/27 01:46:37 INFO fs.TrashPolicyDefault: Namenode trash configuration: Deletion interval=0
minutes, Emptier interval=0 minutes.
Deleted /f2

##新建快照 s3
$hdfs dfs -createSnapshot / s3            //创建快照 s3
Created snapshot /.snapshot/s3            //快照 s3 创建成功

##查看快照 s3 的内容
$hdfs dfs -ls -R /
-rw-r--r--   3 hadoop supergroup     0 2015-01-27 01:37 /f1
drwxr-xr-x   - hadoop supergroup     0 2015-01-06 21:24 /input
-rw-r--r--   3 hadoop supergroup  195257604 2015-01-06 21:24 /input/hadoop-
2.6.0.tar.gz
-rw-r--r--   3 hadoop supergroup    21 2015-01-05 01:47 /input/wordcount
drwxr-xr-x   - hadoop supergroup     0 2015-01-05 01:51 /output
-rw-r--r--   3 hadoop supergroup     0 2015-01-05 01:51 /output/_SUCCESS
-rw-r--r--   3 hadoop supergroup    27 2015-01-05 01:51 /output/part-00000

##比较快照 s2 和 s3 之间的差异
##"+"表示已被创建;"-"表示已被删除;"M"表示已被修改;"R"表示已被重命名
$hdfs snapshotDiff / s2 s3
Difference between snapshot s2 and snapshot s3 under directory /:
M       .
-       ./f2                                //快照 s3 已删除文件 f2
```

通过上述命令可以发现,当前快照 s3 中已经没有 f2 文件,而快照 s2 中还有 f2 文件存在。由此可见,快照只是创建了文件 f2 的一个“逻辑”副本,并没有真正删除 f2 文件。

还可通过 hfds IsSnapshottableDir 和 hdfs dfs -renameSnapshot 命令来进行查看 snapshottable 目录和修改快照名,其具体命令如下。

```
##查看 snapshottable 的目录
$hdfs lsSnapshottableDir                  //一个 snapshottable 目录可以容忍 65536 个 SnapShots
drwxr-xr-x 0 hadoop supergroup 0 2015-01-27 01:47 3 65536 /
##将快照 s1 名改为 s_init
$hdfs dfs -renameSnapshot / s1 s_init

##列出 snapshottable 目录中的所有快照
$hdfs dfs -ls /.snapshot
```




```
Found 3 items
drwxr-xr-x  - hadoop supergroup       0 2015-01-27 01:39 /.snapshot/s2
drwxr-xr-x  - hadoop supergroup       0 2015-01-27 01:47 /.snapshot/s3
drwxr-xr-x  - hadoop supergroup       0 2015-01-27 01:31 /.snapshot/s_init
```

要查看 snapshottable 目录中的所有快照，也可以通过 Web 方式查看，其端口号为 dfs.namenode.http-address 的端口号，如图 4.25 所示。

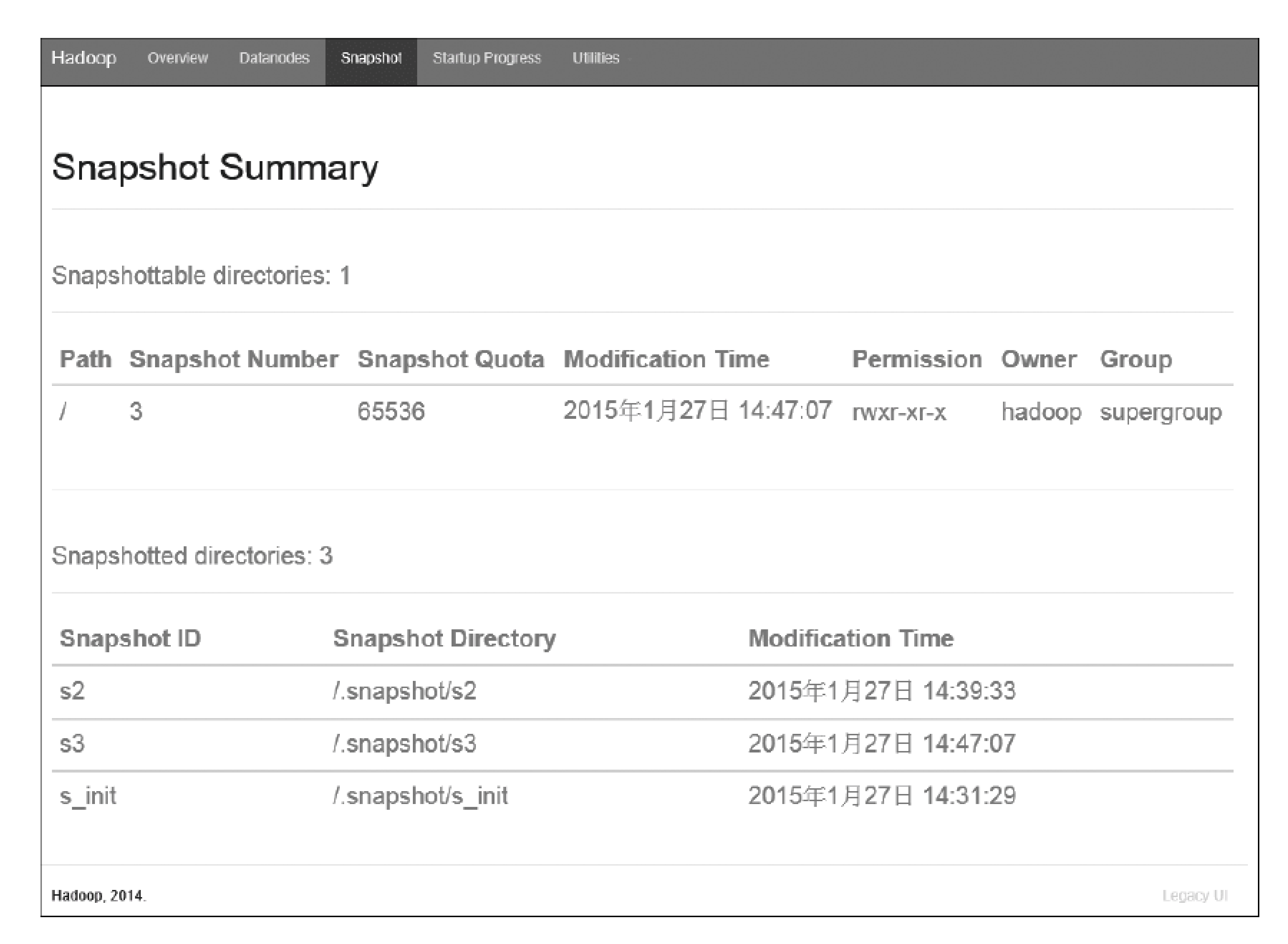


图 4.25　Snapshot Summary

从图 4.25 可以看出，HDFS 的快照共有一个 Snapshottable 目录，并且该目录最多可以容纳 65 536 个 SnapShots 以及一些权限等信息；该目录下有三个 Snapshots，分别为 s2、s3 和 s_init，以及相应的修改时间等信息。

当需要删除快照或关闭 HDFS 快照功能时，可以使用以下命令完成相应操作。

```
##删除快照 s_init、s2 和 s3
$hdfs dfs -deleteSnapshot / s_init
$hdfs dfs -deleteSnapshot / s2
$hdfs dfs -deleteSnapshot / s3

##关闭 Snapshots
$hdfs dfsadmin -disallowSnapshot /
Disallowing snaphot on / succeeded                  //表示根目录快照关闭成功
```

注：有关 HDFS 快照功能的具体实现代码在 org.apache.hadoop.hdfs.server.namenode 包中 INode、INodeDirectory、Snapshot 等类实现，有兴趣的读者可进行深入

阅读。

4.9 HDFS 读写机制

HDFS 是分布式文件系统，要对 HDFS 上的文件进行访问，就需要通过 HDFS 所提供的 API 或 Hadoop dfs/dfsadmin 命令行接口实现与 HDFS 的交互。无论采用哪种方式，其读取或写入 HDFS 数据的机制是相同的，下面将分别对 HDFS 读机制和 HDFS 的写机制进行介绍。

4.9.1 HDFS 读机制

当 Client 端需要读取 HDFS 中的数据时，首先要基于 TCP/IP 与 NameNode 节点建立连接，并通过 Client Protocol 发起读取文件的请求。然后，NameNode 根据用户请求返回相应的块信息。最后，Client 端再向对应块所在的 DataNode 发送请求并取回所需要的数据块，具体的 HDFS 读取机制如图 4.26 所示。

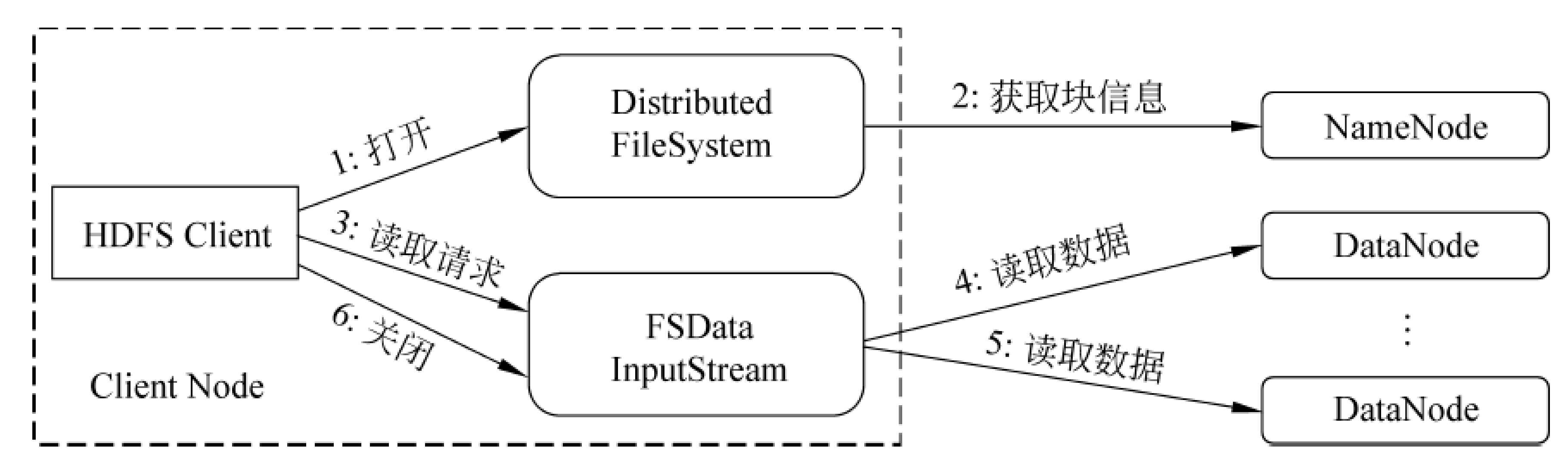


图 4.26 HDFS 读机制

从图 4.26 可以看出，Client 端首先要获取 DistributedFileSystem 的一个实例，并调用该实例的 open()方法，获得该文件对应的输入流 DFSInputStream(FSDataInputStream 继承自 DFSInputStream)，其代码实现如下。

```
public FSDataInputStream open(Path f, final int bufferSize)
      throws IOException {
    statistics.incrementReadOps(1);
    Path absF= fixRelativePart(f);
    return new FileSystemLinkResolver< FSDataInputStream> () {
      @Override
      public FSDataInputStream doCall(final Path p)
          throws IOException, UnresolvedLinkException {
        final DFSInputStream dfsis=
          dfs.open(getPathName(p), bufferSize, verifyChecksum);
        return dfs.createWrappedInputStream(dfsis);
      }
      @Override
```

```
        public FSDataInputStream next(final FileSystem fs, final Path p)
            throws IOException {
          return fs.open(p, bufferSize);
        }
      }.resolve(this, absF);
    }
```

其中，dfs 为 DistributedFileSystem 的成员变量 DFSClient。当 open()函数被调用后，创建输入流 DFSInputStream。在 DFSInputStream 的构造函数中，openInfo 函数被调用，该函数通过 RPC 远程调用 NameNode 获得该文件对应的数据块相关信息，其代码实现如下。

```
  synchronized void openInfo() throws IOException, UnresolvedLinkException {
      lastBlockBeingWrittenLength= fetchLocatedBlocksAndGetLastBlockLength();
      int retriesForLastBlockLength= dfsClient.getConf().
      retryTimesForGetLastBlockLength;
      while(retriesForLastBlockLength > 0) {
        if(lastBlockBeingWrittenLength= = - 1) {
          DFSClient.LOG.warn("Last block locations not available. "
              + "Datanodes might not have reported blocks completely."
              + " Will retry for "+ retriesForLastBlockLength+ " times");
          waitFor(dfsClient.getConf().retryIntervalForGetLastBlockLength);
          lastBlockBeingWrittenLength= fetchLocatedBlocksAndGetLastBlockLength();
        } else {
          break;
        }
        retriesForLastBlockLength- - ;
      }
      if(retriesForLastBlockLength= = 0) {
        throw new IOException("Could not obtain the last block locations.");
      }
    }
```

Client 端获得 DFSInputStream 之后，调用 read()方法读取数据块。输入流 DFSInputStream 会根据排序结果，选择最近的 DataNode 建立连接并读取数据，如果 Client 端与其中的一个 DataNode 位于同一个节点，则直接从该节点读取数据。DFSInputStream. read()方法代码实现如下。

```
  public int read(long position, byte[] buffer, int offset, int length)
      throws IOException {
      synchronized(this) {
        long oldPos= getPos();
        int nread= - 1;
```

```
        try {
          seek(position);
          nread= read(buffer, offset, length);
        } finally {
          seek(oldPos);
        }
        return nread;
      }
    }
```

如果已读取到数据块末端,则关闭与该 DataNode 的连接,然后重新查找下一个数据块,直到该文件的所有数据块全部读完为止。最后,Client 端调用 close()方法,关闭输入流 DFSInputStream,完成对 HDFS 中文件的读操作。其中,close()方法的代码实现如下。

```
  public synchronized void close()throws IOException {
      if(closed){
        return;
      }
      dfsClient.checkOpen();

      if(!extendedReadBuffers.isEmpty()){
        final StringBuilder builder= new StringBuilder();
        extendedReadBuffers.visitAll(new IdentityHashStore.Visitor<ByteBuffer, Object>(){
          private String prefix= "";
          @Override
          public void accept(ByteBuffer k, Object v){
            builder.append(prefix).append(k);
            prefix= ", ";
          }
        });
        DFSClient.LOG.warn("closing file "+ src+ ", but there are still "+
            "unreleased ByteBuffers allocated by read().  "+
            "Please release "+ builder.toString()+ ".");
      }
      if(blockReader != null){
        blockReader.close();
        blockReader= null;
      }
      super.close();
      closed= true;
    }
```

如果 DFSInputStream 和 DataNode 的通信发生异常，或者数据校验出错，DFSInputStream 就会重新选择 DataNode 传输数据。

4.9.2 HDFS 写机制

当 Client 端在读 HDFS 的数据时，对文件的操作包括 open()、read()、close()等过程，当 Client 端在写入数据到 HDFS 时，其操作主要包括 create()、write()、close()等过程，具体的 HDFS 写机制如图 4.27 所示。

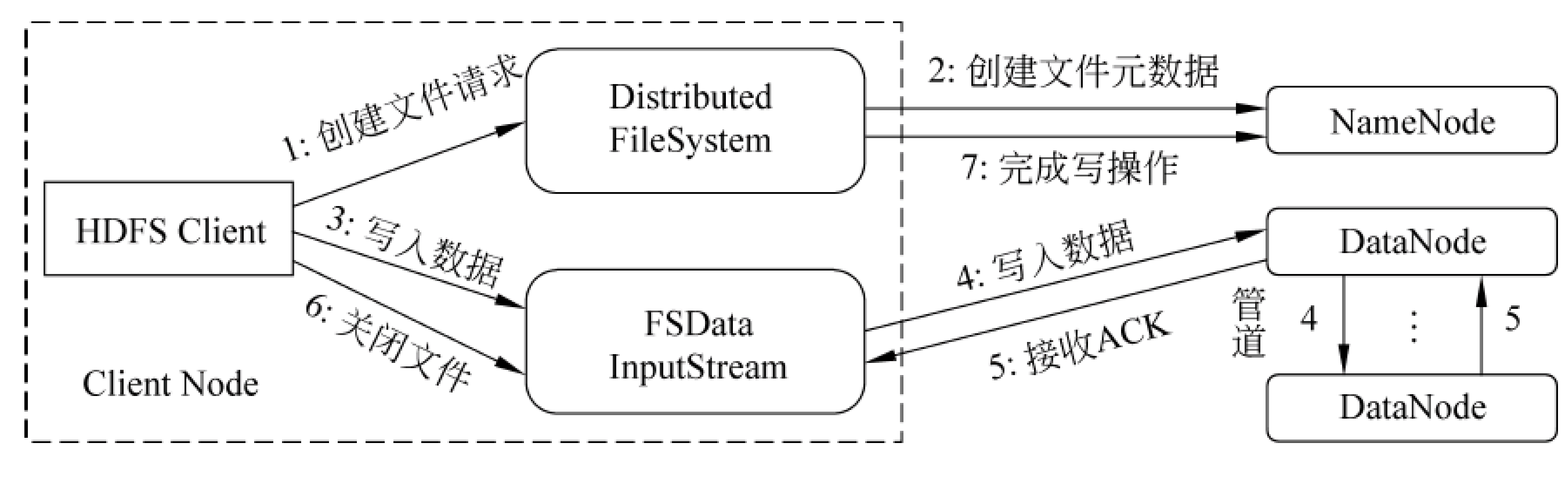


图 4.27　HDFS 写机制

从图 4.27 可以看出，Client 端首先要获取 DistributedFileSystem 的一个实例，并调用该实例的 create()方法，创建文件，其代码实现如下。

```
public FSDataOutputStream create(Path f, FsPermission permission,
        boolean overwrite, int bufferSize, short replication, long blockSize,
        Progressable progress) throws IOException {
    return this.create(f, permission,
            overwrite ?EnumSet.of(CreateFlag.CREATE, CreateFlag.OVERWRITE)
                : EnumSet.of(CreateFlag.CREATE), bufferSize, replication,
            blockSize, progress, null);
  }
```

DistributedFileSystem.create()方法返回一个 FSDataOutputStream 用于向新生成的文件中写入数据，其成员变量 dfs 的类型为 DFSClient，调用 DFSClient.create()方法，构造了一个 DFSOutputStream，并在构造函数中通过 RPC 调用 NameNode 的 create()方法来创建一个文件。NameNode 检查文件是否已存在、操作权限。如果检查通过，NameNode 记录新文件信息，并在某一个 DataNode 上创建数据块，并返回 FSDataOutputStream，将 Client 引导至该数据块执行写入操作。Client 端调用 FSDataOutputStream.write()方法向 HDFS 中对应的文件写入数据，其真正写数据的操作是由 DFSOutputStream.wirteChunk()实现的，其代码实现如下。

```
protected synchronized void writeChunk(byte[] b, int offset, int len,byte[] checksum, int ckoff, int
cklen)throws IOException {
    dfsClient.checkOpen();
```

```
checkClosed();
if(len > bytesPerChecksum){
  throw new IOException("writeChunk() buffer size is "+ len+" is larger than supported
  bytesPerChecksum "+bytesPerChecksum);
}
if(cklen != 0 && cklen != getChecksumSize()){
  throw new IOException("writeChunk() checksum size is supposed to be "+getChecksumSize()+" but
  found to be "+cklen);
}
//创建一个 package,并写入数据
if(currentPacket==null){
  currentPacket=createPacket(packetSize, chunksPerPacket, bytesCurBlock, currentSeqno++);
  if(DFSClient.LOG.isDebugEnabled()){
    DFSClient.LOG.debug("DFSClient writeChunk allocating new packet seqno="+
        currentPacket.seqno+
        ", src="+src+
        ", packetSize="+packetSize+
        ", chunksPerPacket="+chunksPerPacket+
        ", bytesCurBlock="+bytesCurBlock);
  }
}
currentPacket.writeChecksum(checksum, ckoff, cklen);
currentPacket.writeData(b, offset, len);
currentPacket.numChunks++;
bytesCurBlock+=len;
//如果此 package 已满,则放入队列中准备发送
if(currentPacket.numChunks==currentPacket.maxChunks ||
    bytesCurBlock==blockSize){
  if(DFSClient.LOG.isDebugEnabled()){
    DFSClient.LOG.debug("DFSClient writeChunk packet full seqno=
    "+ currentPacket.seqno+", src="+src+", bytesCurBlock="+bytesCurBlock+", blockSize="
    +blockSize+", appendChunk="+appendChunk);
  }
  waitAndQueueCurrentPacket();
  if(appendChunk && bytesCurBlock%bytesPerChecksum==0){
    appendChunk=false;
    resetChecksumBufSize();
  }
  if(!appendChunk){
    {int psize = Math.min((int)(blockSize - bytesCurBlock), dfsClient.getConf().
     writePacketSize);
    computePacketChunkSize(psize, bytesPerChecksum);
```

```
            }
            if(bytesCurBlock==blockSize){
              currentPacket=createPacket(0, 0, bytesCurBlock, currentSeqno++);
              currentPacket.lastPacketInBlock=true;
              currentPacket.syncBlock=shouldSyncBlock;
              waitAndQueueCurrentPacket();
              bytesCurBlock=0;
              lastFlushOffset=0;
            }
          }
        }
```

当 Client 端开始写入文件的时候，文件将被切分成多个 Packages，在内部以数据队列（Data Queue）形式管理这些 Packages。DFSOutputStream 还有一个队列叫 ACK Queue，等待 DataNode 的收到响应，当管道 Pipline 中的所有 DataNode 都表示已经收到的时候，ACK Queue 才会把对应的 Packages 移除掉。

当 Client 端完成写数据后，调用 close()方法关闭写入流，此时 Client 端将不会再向流中写入数据。当 DFSOutputStream 数据队列的文件包全部收到 ACK 应答后，通知 NameNode 关闭文件，完成一次正常的文件写入。如果在写的过程中某个 DataNode 发生错误，数据流管道会被关闭，已经发送到管道但是还没有收到确认的文件包，会被重新添加到 DFSOutputStream 的输出队列，重新建立新的管道写数据到正常的 DataNode 中。

4.10　HDFS 常用操作

HDFS 提供了 dfs 命令行接口、dfsadmin 命令行接口、Web 接口及 HDFS API 等方式与 HDFS 文件系统进行交互。用户可以很方便地进行上传文件、读/写文件、创建目录、重命名、删除文件等操作。

4.10.1　dfs 命令

Hadoop 包括多种 Shell 风格的命令，用于跟 HDFS 或者 Hadoop 支持的其他文件系统交互。其中，hdfs dfs 命令可以列出 Hadoop Shell 支持的命令，只能支持 HDFS 文件系统相关的操作，hdfs dfs 具体命令如下所示。

```
[hadoop@master1 ~]$ hdfs dfs
Usage: hadoop fs [generic options]
        [-appendToFile <localsrc> ... <dst>]
        [-cat [-ignoreCrc] <src> ...]
        [-checksum <src> ...]
        [-chgrp [-R] GROUP PATH...]
```

```
        [-chmod [-R] <MODE[,MODE]... | OCTALMODE> PATH...]
        [-chown [-R] [OWNER][:[GROUP]] PATH...]
        [-copyFromLocal [-f] [-p] [-l] <localsrc> ... <dst>]
        [-copyToLocal [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-count [-q] [-h] <path> ...]
        [-cp [-f] [-p | -p[topax]] <src> ... <dst>]
        [-createSnapshot <snapshotDir> [<snapshotName>]]
        [-deleteSnapshot <snapshotDir> <snapshotName>]
        [-df [-h] [<path> ...]]
        [-du [-s] [-h] <path> ...]
        [-expunge]
        [-get [-p] [-ignoreCrc] [-crc] <src> ... <localdst>]
        [-getfacl [-R] <path>]
        [-getfattr [-R] {-n name | -d} [-e en] <path>]
        [-getmerge [-nl] <src> <localdst>]
        [-help [cmd ...]]
        [-ls [-d] [-h] [-R] [<path> ...]]
        [-mkdir [-p] <path> ...]
        [-moveFromLocal <localsrc> ... <dst>]
        [-moveToLocal <src> <localdst>]
        [-mv <src> ... <dst>]
        [-put [-f] [-p] [-l] <localsrc> ... <dst>]
        [-renameSnapshot <snapshotDir> <oldName> <newName>]
        [-rm [-f] [-r|-R] [-skipTrash] <src> ...]
        [-rmdir [--ignore-fail-on-non-empty] <dir> ...]
        [-setfacl [-R] [{-b|-k} {-m|-x <acl_spec>} <path>]|[--set <acl_spec> <path>]]
        [-setfattr {-n name [-v value] | -x name} <path>]
        [-setrep [-R] [-w] <rep> <path> ...]
        [-stat [format] <path> ...]
        [-tail [-f] <file>]
        [-test -[defsz] <path>]
        [-text [-ignoreCrc] <src> ...]
        [-touchz <path> ...]
        [-usage [cmd ...]]

Generic options supported are
-conf <configuration file>     specify an application configuration file
-D <property=value>            use value for given property
-fs <local|namenode:port>      specify a namenode
-jt <local|resourcemanager:port>    specify a ResourceManager
-files <comma separated list of files>    specify comma separated files to be copied to the map
reduce cluster
```

```
-libjars <comma separated list of jars>      specify comma separated jar files to include in the
classpath.
-archives <comma separated list of archives>     specify comma separated archives to be unarchived
on the compute machines.

The general command line syntax is
bin/hadoop command [genericOptions] [commandOptions]
```

可以看出，这些命令不仅支持一般文件系统的操作，例如复制文件、修改文件权限等，同时也支持了部分 HDFS 特有的命令，例如修改文件的 replication 因子。而且，hdfs dfs 相当于 hadoop fs，两者的区别在于 hadoop fs 可以操作任何文件系统，而 hdfs dfs 只能操作 HDFS 文件系统。读者可使用 hdfs dfs -help 命令展现特定命令的帮助细节，并对 HDFS 进行操作。

4.10.2　dfsadmin 命令

hdfs dfsadmin 命令支持一些 HDFS 管理功能的操作。hdfs dfsadmin -help 命令可以列出所有当前支持的命令，hdfs dfsadmin 具体命令如下所示。

```
[hadoop@master1 ~]$ hdfs dfsadmin
Usage: hdfs dfsadmin
Note: Administrative commands can only be run as the HDFS superuser.
        [-report [-live] [-dead] [-decommissioning]]
        [-safemode <enter | leave | get | wait>]
        [-saveNamespace]
        [-rollEdits]
        [-restoreFailedStorage true|false|check]
        [-refreshNodes]
        [-setQuota <quota> <dirname>...<dirname>]
        [-clrQuota <dirname>...<dirname>]
        [-setSpaceQuota <quota> <dirname>...<dirname>]
        [-clrSpaceQuota <dirname>...<dirname>]
        [-finalizeUpgrade]
        [-rollingUpgrade [<query|prepare|finalize>]]
        [-refreshServiceAcl]
        [-refreshUserToGroupsMappings]
        [-refreshSuperUserGroupsConfiguration]
        [-refreshCallQueue]
        [-refresh <host:ipc_port> <key> [arg1..argn]]
        [-reconfig <datanode|...> <host:ipc_port> <start|status>]
        [-printTopology]
        [-refreshNamenodes datanode_host:ipc_port]
```

```
            [-deleteBlockPool datanode_host:ipc_port blockpoolId [force]]
            [-setBalancerBandwidth <bandwidth in bytes per second>]
            [-fetchImage <local directory>]
            [-allowSnapshot <snapshotDir>]
            [-disallowSnapshot <snapshotDir>]
            [-shutdownDatanode <datanode_host:ipc_port> [upgrade]]
            [-getDatanodeInfo <datanode_host:ipc_port>]
            [-metasave filename]
            [-setStoragePolicy path policyName]
            [-getStoragePolicy path]
            [-help [cmd]]

    Generic options supported are
    -conf <configuration file>      specify an application configuration file
    -D <property=value>              use value for given property
    -fs <local|namenode:port>       specify a namenode
    -jt <local|resourcemanager:port>     specify a ResourceManager
    -files <comma separated list of files>     specify comma separated files to be copied to the map
    reduce cluster
    -libjars <comma separated list of jars>     specify comma separated jar files to include in the
    classpath.
    -archives <comma separated list of archives>     specify comma separated archives to be unarchived
    on the compute machines.

    The general command line syntax is
    bin/hadoop command [genericOptions] [commandOptions]
```

可以看出，hdfs dfsadmin 命令相当于 hadoop dfsadmin 命令，例如：hdfs/hadoop dfsadmin -report 命令报告 HDFS 的基本统计信息，部分信息同时展现在 NameNode 的 Web 首页上。读者可使用 hdfs dfs -help 命令展现特定命令的帮助细节，并对 HDFS 进行管理操作。

4.10.3 Web 接口

NameNode 和 DataNode 分别内置了一个 Web 服务器，来展现集群当前状态的一些基本信息。在默认配置下，NameNode 的首页地址是 http://namenode:50070（NameNode 就是 NameNode 节点所在机器 IP 或者名称）。例如，本实例 Master1.Hadoop 节点为 NameNode 节点，其 IP 地址为 192.168.85.100，则 NameNode 的首页地址为 http://192.168.85.100:50070，如图 4.28 所示。

这个页面列出了集群中的所有 DataNode 以及集群的基本统计。Web 接口同样可以用于浏览文件系统，可以单击 NameNode 首页上的 Utilities→Browse the file system 链

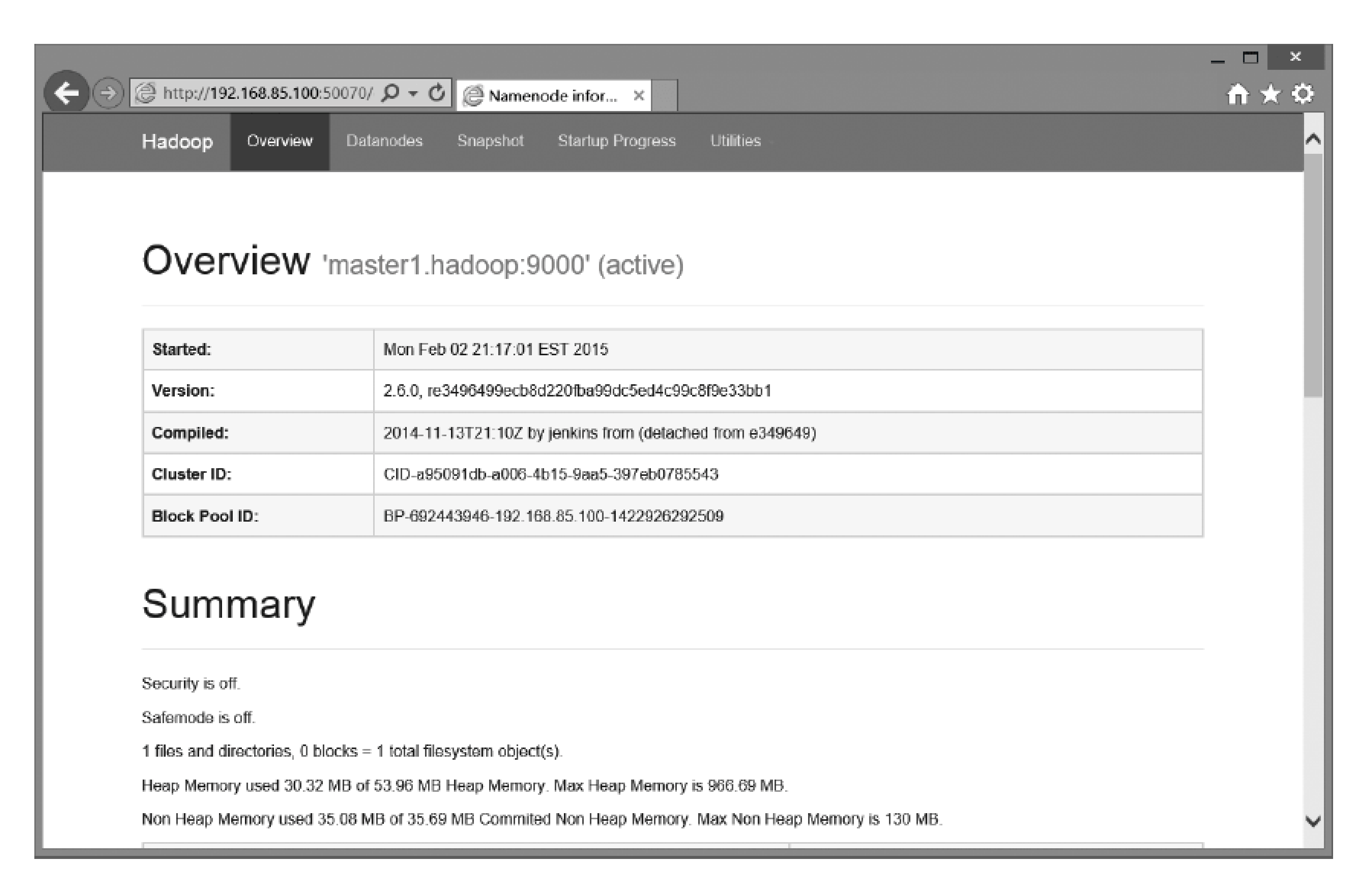


图 4.28　集群基本信息统计

接，出现如图 4.29 所示的界面。

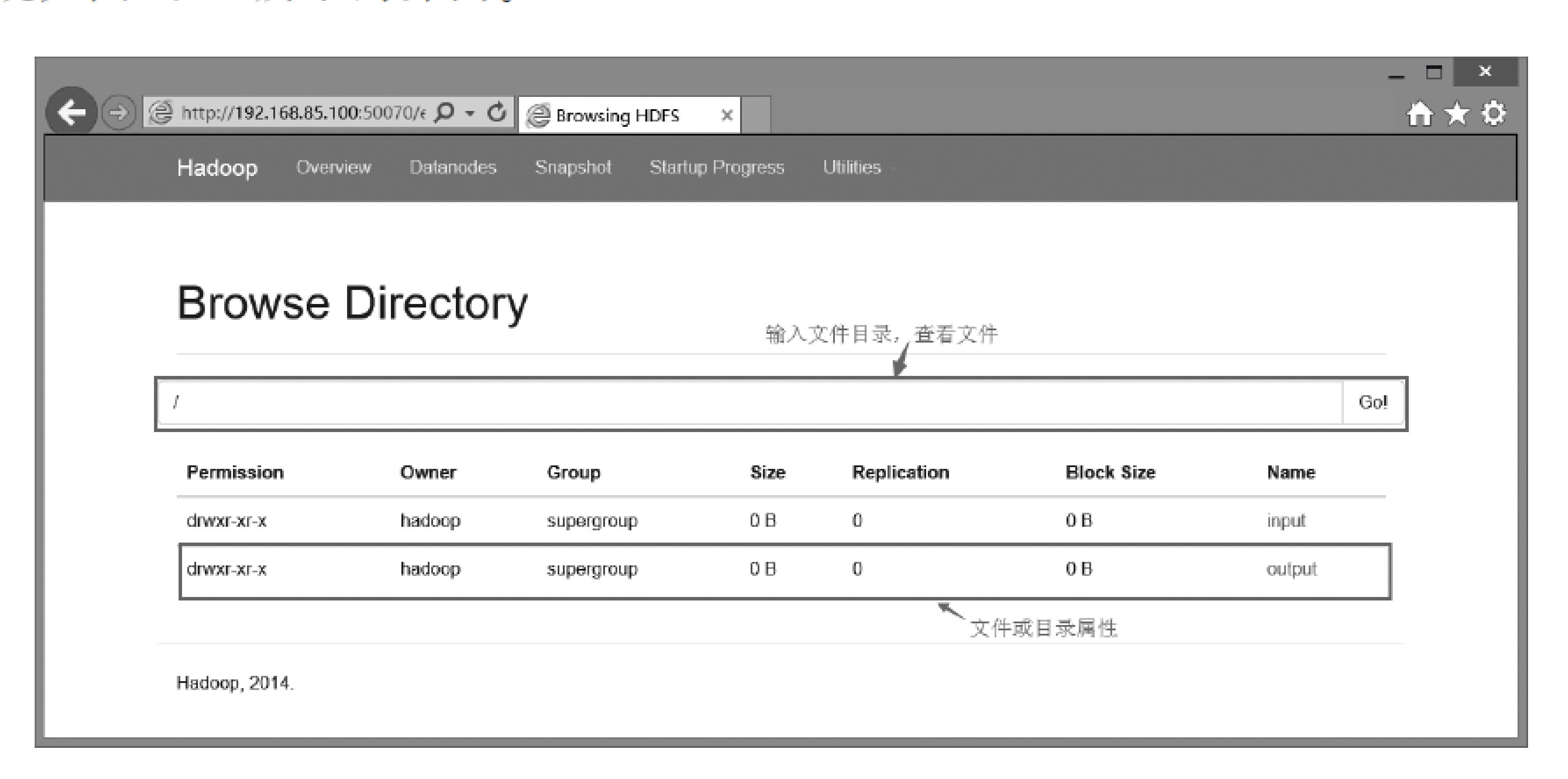


图 4.29　HDFS 文件系统 Web 接口

该页面给出了 HDFS 文件系统的目录及文件结构，可以输入目录，查看该目录下的文件或目录属性。本实例在 HDFS 的根目录“/”下有两个目录，分别为 input 目录和 output 目录。我们可以进入 input 目录，该目录下有 wordcount 文件，单击该文件可以查看在 HDFS 上的存储信息，也可以把该文件下载到本地，如图 4.30 所示。

通过 Web 接口方式访问 HDFS，只能对文件或目录进行查操作，无法对 HDFS 上的文件进行读写操作。如要对 HDFS 上的文件进行读写操作，可使用 dfs 命令或 HDFS API 的方式。

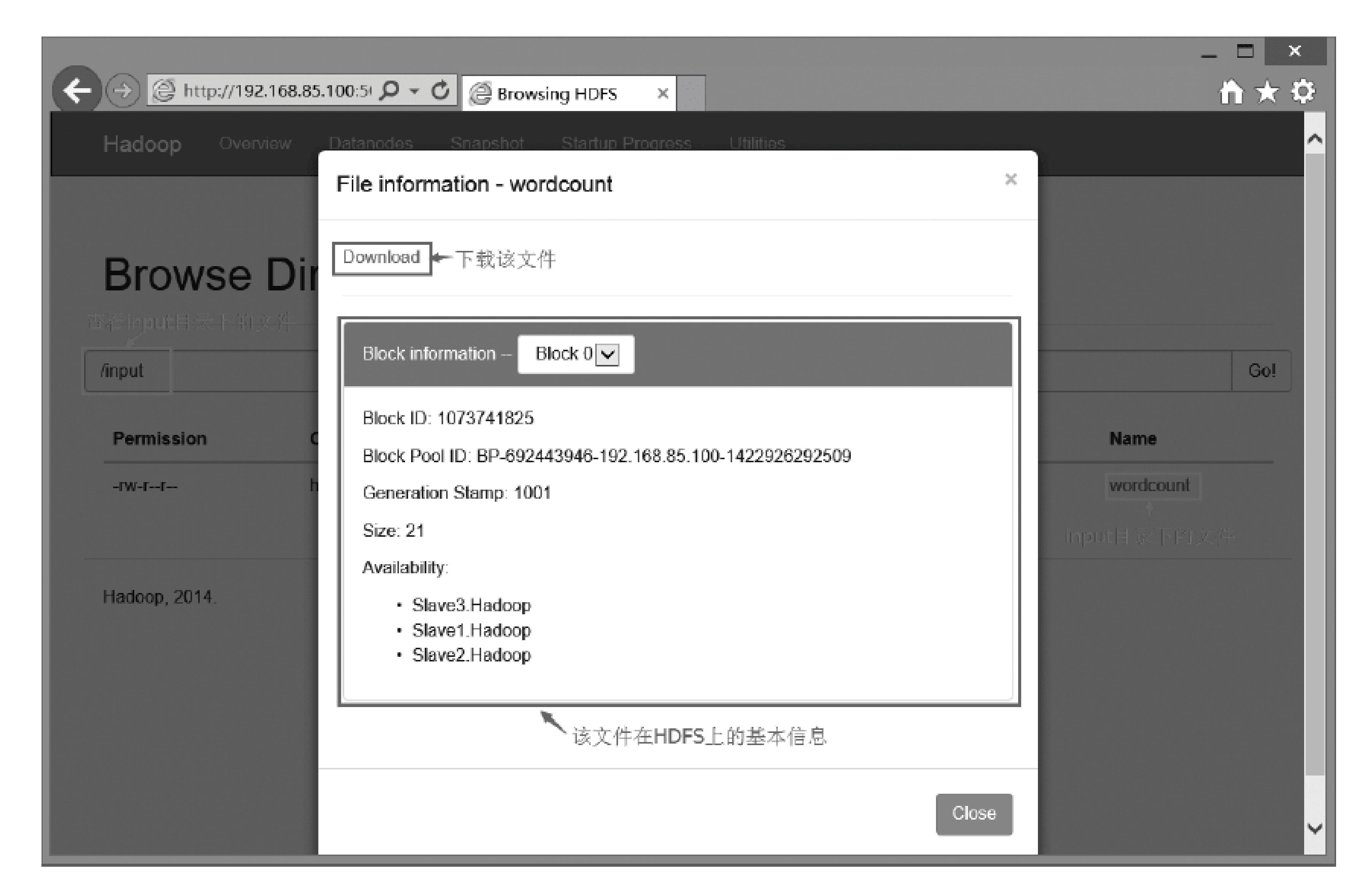


图 4.30 HDFS 文件目录查看

4.10.4 HDFS API

HDFS API 可以对 HDFS 中的文件进行新建文件、读/写文件内容、目录创建、重命名和删除文件等操作，这些操作主要涉及 HDFS API 的 Configuration 类、FlieSystem 类、FSDataInputStream 类和 FSDataOutputStream 类。其中，Configuration 类主要用于封装客户端和服务器端的配置；FileSystem 类提供对文件的操作方法；FSDataInputStream 类和 FSDataOutputStream 类是 HDFS 的输入输出流，分别通过 FlieSystem. open()方法和 FileSystem. create()方法获得。下面将介绍通过 HDFS API 对文件进行各种操作。

1. 读取文件

文件在 HDFS 中显示为一个 Path 对象或者视为 HDFS 的一个 URI，可通过 FileSystem. get()方法返回一个访问文件的指针，然后打开文件进行数据读取。本实例将读取 HDFS 中 input 目录下的 wordcout 文件内容（请查看本书配套资料 HadoopWorksapces 文件夹下的 HDFS 项目）。首先创建一个 MapRedue 工程 Map/Reduce Project（有关 MapReduce 工程的创建请查看第 5 章的相关内容），新建一个 Java 类（ReadDataFromHDFS），其代码内容如下。

```
import java.io.IOException;
import java.net.URI;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataInputStream;
```

```
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

public class ReadDataFromHDFS {
    public static void main(String[] args)throws Exception, IOException {
        String uri= "hdfs://master1.hadoop:9000/input/wordcount";
        Configuration conf=new Configuration();
        FileSystem fs=FileSystem.get(URI.create(uri), conf);
        FSDataInputStream in=null;
        try {
            in=fs.open(new Path(uri));
            IOUtils.copyBytes(in, System.out, 4096, false);
        } finally {
            IOUtils.closeStream(in);
        }
    }
}
```

FileSystem.get()方法用于创建一个 DistributedFileSystem 类的对象，然后通过该对象按照 HDFS 的 API 对 HDFS 中的文件和目录进行操作，如读取 HDFS 中 input 目录下的 wordcount 文件，即 uri="hdfs://master1.hadoop:9000/input/wordcount"。单击 Run As→Run on Hadoop，输出的结果为（HDFS 中 wordcount 文件的内容）：

```
Hello World!
Hadoop!
```

2. 写入文件

本实例将在 HDFS 的 input 目录下创建一个名为“in1”的文件并写入“Hello World!”字符串（请查看本书配套资料 HadoopWorksapces 文件夹下的 HDFS 项目中的 WriteDataToHDFS 类），其代码内容如下。

```
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FSDataOutputStream;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class WriteDataToHDFS {
    public static void main(String[] args)throws Exception {
        Configuration conf=new Configuration();
```

```
            String dstFile= "hdfs://master1.hadoop:9000/input/in1";
            Path dstPath= new Path(dstFile);
            FileSystem fs= FileSystem.get(URI.create(dstFile), conf);
            byte[] buff= "Hello World!".getBytes();
            FSDataOutputStream outputStream= fs.create(dstPath);
            outputStream.write(buff, 0, buff.length);
        }
    }
```

FileSystem.create()方法返回的输出流,可以对指定的 HDFS 上的文件进行数据的写入操作,用 FSDataOutputStream.write()方法写入。最后,在 HDFS 中的 input 目录下将生成一个名为“in1”的文件,其文件内容为“Hello World!”,运行结果如图 4.31 所示。



图 4.31 写入文件的运行结果

3. 上传文件

本实例将把本地目录/home/hadoop 下的 wordcount 文件上传到 HDFS 的根目录下(请查看本书配套资料 HadoopWorksapces 文件夹下的 HDFS 项目中的 UploadFileToHDFS 类),其代码内容如下。

```
import java.io.IOException;
import java.net.URI;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;

public class UploadFIleToHDFS {
    public static void main(String[] args)throws IOException {
        Configuration conf= new Configuration();
        String srcFile= "/home/hadoop/wordcount";
        String dstFile= "hdfs://master1.hadoop:9000/";
        FileSystem fs= FileSystem.get(URI.create(dstFile), conf);
        Path srcPath= new Path(srcFile);
```

```
            Path dstPath=new Path(dstFile);
            fs.copyFromLocalFile(srcPath, dstPath);
        }
    }
```

在上传本地文件到 HDFS 时，会用到 FileSystem.copyFromLocalFile()方法，其中 srcPath 和 dstPath 都必须为完整路径。该程序的运行结果如图 4.32 所示。

Hadoop　Overview　Datanodes　Snapshot　Startup Progress　Utilities

Browse Directory

/　Go!

Permission	Owner	Group	Size	Replication	Block Size	Name
drwxr-xr-x	hadoop	supergroup	0 B	0	0 B	input
drwxr-xr-x	hadoop	supergroup	0 B	0	0 B	output
-rw-r--r--	hadoop	supergroup	21 B	3	128 MB	wordcount

上传成功

图 4.32　上传文件结果

有关通过 HDFS API 对 HDFS 的更多操作，请查看 Hadoop API(请查看本书配套资料中 Hadoop-2.6.0-API 文件夹中的相关内容)。其中，FileSystem 类提供了对 HDFS 中文件的很多操作方法，比如：

(1) listStatus()方法查看指定 HDFS 目录下的文件列表；

(2) create()方法用于在 HDFS 上创建文件；

(3) mkdir()方法用于在 HDFS 上创建目录；

(4) rename()方法用于在 HDFS 上对文件进行重命名；

(5) delete()方法用于在 HDFS 上对指定文件进行删除操作；

(6) getmodificationtime()方法用于查看 HDFS 上指定文件的修改时间；

(7) exits()方法用于查看指定的 HDFS 文件是否存在；

(8) getBlockLocation()方法用于查看指定 HDFS 文件在集群中的位置；

(9) getHostname()方法用于获取 HDFS 集群上所有的节点名称。

请读者认真阅读有关 HDFS API 的相应类，如 Configuration 类、FlieSystem 类、FSDataInputStream 类和 FSDataOutputStream 类，并实际操作进行验证和理解。

第5章 分布式计算框架 MapReduce

MapReduce 是一种用于大规模数据处理的分布式计算框架，是 Hadoop 的核心组件之一。它的主要思想来自于分而治之(Map 过程和 Reduce 过程)，Map 过程是将一个大的问题切换成很多小的问题，然后在集群中的各个节点上执行，Reduce 过程是将 Map 阶段产生的结果进行汇集。本章将重点介绍 MapReduce 的架构结构、工作原理以及实例演示等内容。

5.1 MapReduce 概述

MapReduce[66]是一种高性能的批处理分布式计算框架，用于海量数据的并行分析和处理。该框架的基本原理和主要设计思想来源于 2004 年 Google 公司 Dean Jeffrey 等人在国际会议上发表的 *MapReduce: Simplified Data Processing on Large Clusters* 论文[66]。同年，开源项目 Lucene(搜索索引程序库)和 Nutch(搜索引擎)的创始人 Doug Cutting 为了解决搜索引擎中大规模网页数据的并行化处理，实现了基于 Google MapReduce 思想的开源 MapReduce。目前，MapReduce 已成为 Hadoop 子项目中最重要的项目之一，而且被广泛应用于海量数据的搜索、挖掘、分析等方面。MapReduce 的主要特点如下。

1. 可扩展性

MapReduce 的扩展性包括数据的可扩展性和计算规模的可扩展性。MapReduce 的数据可扩展性体现在随着数据规模的扩大而表现持续的有效性；计算规模的可扩展性体现在随着集群中节点数的增加保持接近线性的增长。

2. 高容错性

MapReduce 的高容错性体现在可动态增加/减少计算节点，真正实现了弹性计算，而且该框架使用了多种有效的机制来提高容错性，如节点自动重启技术，当集群中的节点失效时，能有效处理失效节点的检测和恢复。

3. 高效性

MapReduce 采用了数据/代码互定位的技术方法，减少了集群中的数据通信，从而有

效降低网络带宽。集群中的每个计算节点尽量负责处理本地存储的数据，仅当本地节点无法处理本地数据时，再采用就近原则寻找其他可用计算节点，并把数据传送到该可用计算节点。

4. 动态灵活的资源分配和调度

MapReduce 将提交的计算作业分为多个计算任务，任务调度功能主要负责为这些划分后的计算任务分配和调度计算节点(Map 节点和 Reduce 节点)，同时也负责一些计算性能的优化处理，如支持作业调度的优先级和任务抢占等。

5. 隐藏底层细节

MapReduce 的最大优势在于封装了底层实现细节，减少了编程人员在大规模数据处理时需要考虑诸如数据分布存储管理、数据分发、数据通信和同步等诸多细节问题，有效降低了并行编程难度。编程人员可以从底层细节中解放出来，仅需要关心应用层本身的算法设计，从而提高了编程人员在分布式编程环境下的编程效率。

总之，MapReduce 提供了一个统一的计算框架，实现了在分布式环境中计算任务的划分和调度，数据的分布存储和划分，处理数据与计算任务的同步，结果数据的收集整理，计算性能优化处理，计算节点出错检测和失效恢复等功能。MapReduce 适合处理各种类型的数据，包括结构化、半结构化和非结构化数据，而且处理的数据量在 TB 和 PB 级别，并且经过大量实际生产环境的使用和验证，具有高可用性。

5.2　MapReduce 原理

MapReduce 的开发和实现遵循了 Google MapReduce 的设计思想，即 MapReduce 将任务分解为大量的并行 Map 任务和 Reduce 汇总任务。然后把拆分成的若干个 Map 任务分配到不同的节点上去执行，每一个 Map 任务处理输入数据中的一部分。当 Map 任务完成之后，会生成一些中间文件，而这些中间文件将会作为 Reduce 任务的输入数据。各个 Reduce 任务把前面若干个 Map 任务完成后的中间结果进行汇总，最终汇总所有 Reduce 任务的输出结果即可获得最终结果。简单地说，MapReduce 就是任务的分解与结果的汇总，其处理过程如图 5.1 所示。

其中，每个 Map 或 Reduce 任务都相对独立，而且在进行 Reduce 任务之前，必须等到所有的 Map 任务执行完成并输出结果。在 Hadoop 中，每个 MapReduce 任务都被初始化为一个作业(Job)，每个作业又可以分为 Map 阶段和 Reduce 阶段，这两个阶段分别由 map 函数和 reduce 函数表示。其中，map 函数用于接收基于＜key，value＞形式的数据集，然后同样产生＜key，value＞形式的中间输出；reduce 函数用于接收基于＜key，(list of values)＞形式的输入数据，然后对 value 集合进行处理，每个 reduce 产生 0 个或 1 个输出，其输出的形式也为＜key，value＞。在此过程中，Map 阶段和 Reduce 阶段之间还有一系列的过程，即把 Map 执行完成产生的中间结果＜key，value＞转化为 Reduce 任务的输入数据＜key，(list of values)＞，如分区(Partition)、排序(Sort)、压缩(Combine)、复制

(Copy)、合并(Merge)等过程,这些过程往往被统称为"Group 阶段",其目的就是对数据进行梳理、排序,以更科学的方式分发给每个 Reduce 阶段,以便能够更高效地进行计算和处理。

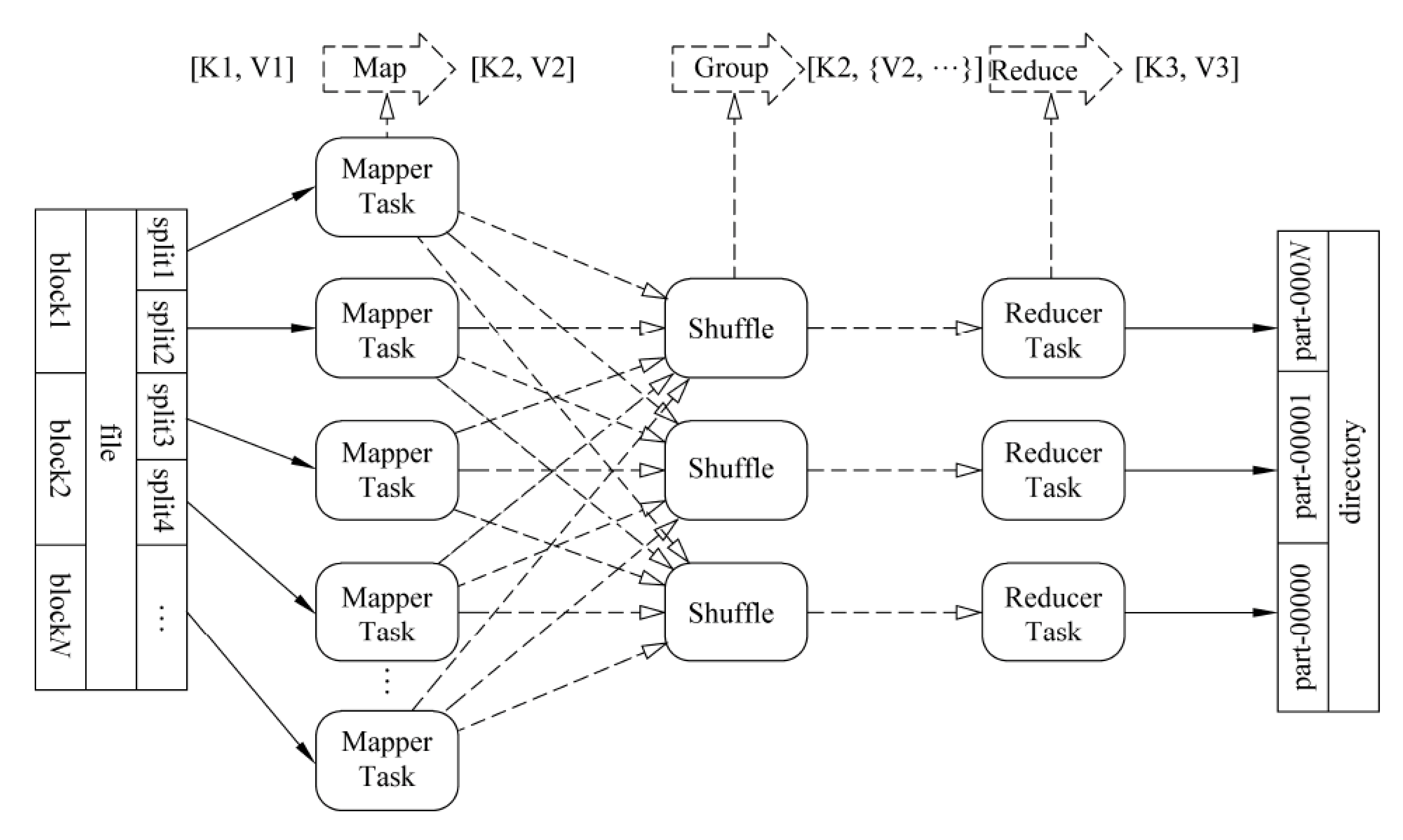


图 5.1 MapReduce 原理

5.3 MapReduce 框架

MapReduce 的基本工作原理比较简单,但要开发一套具有可扩展性、可用性、灵活性的 MapReduce 框架并不容易,不仅要考虑到 Map 和 Reduce 并行计算本身,还要考虑到并行编程中分布式存储、工作调度、负载均衡、容错均衡、容错处理及网络通信等复杂问题。下面将介绍在 Hadoop 1.0 生态系统和 Hadoop 2.0 生态系统中的 MapReduce 框架。

5.3.1 Hadoop 1.0 生态系统中 MapReduce 框架

在 Hadoop 1.0 生态系统中的 MapReduce 框架采用 Master/Slave 架构模式,并且用于执行 MapReduce 任务的机器角色有两个,即 Master JobTracker 和 Slave TaskTracker,如图 5.2 所示。这两个角色和分布式文件系统 HDFS 的角色运行在一组相同的节点上,即计算节点和存储节点在一起。其中,Master JobTracker 负责调度构成一个作业的所有任务,这些任务分布在不同的 Slave 节点上;Slave TaskTracker 仅负责执行由 Master 指派的任务。Hadoop 1.0 生态系统中的 MapReduce 框架(MRv1)如图 5.2 所示。

从图 5.2 中可以看出,MapReduce 框架实现了一个相对简单的集群管理器来执行 MapReduce 处理,其工作流程如下。

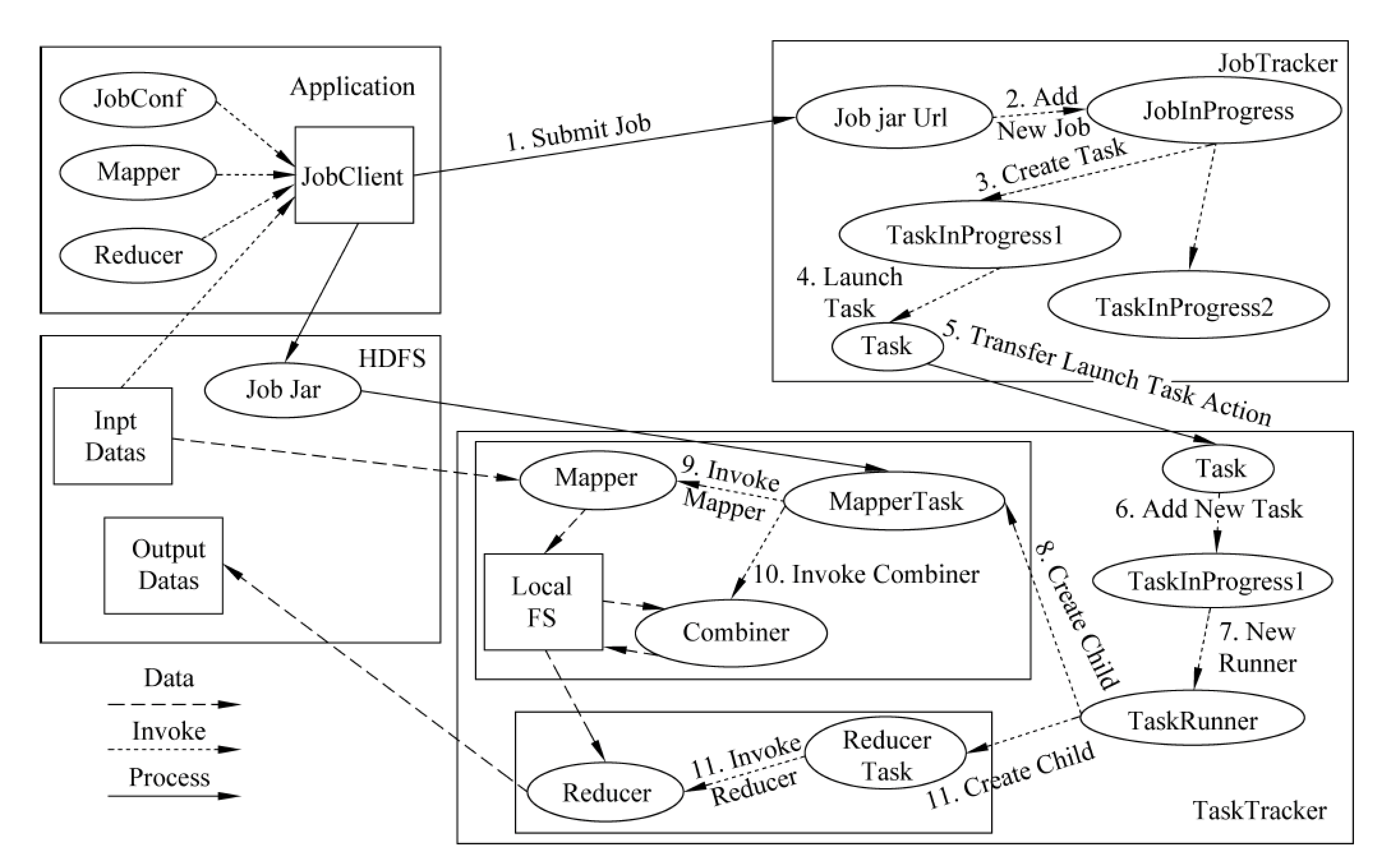


图 5.2　Hadoop 1.0 生态系统中的 MapReduce 框架

（1）当一个 Client 端向一个 Hadoop 集群发出一个作业请求时，由用户程序(JobClient)提交一个作业(Job)，该作业的信息会发送到 JobTracker 中。

（2）JobTracker 主要负责与集群中的节点定时通信，并将 Map 和 Reduce 任务安排到一个或多个 TaskTracker 上的可用节点中。此外，JobTracker 还负责所有 Job 的重启和失败等操作。

（3）TaskTracker 主要负责监视自己所在节点的资源情况，并监视当前节点的 tasks 运行状况。同时，TaskTracker 需要把这些信息通过心跳机制发送给 JobTracker，JobTracker 根据这些信息为新提交的 Job 分配运行节点。图中虚线箭头表示消息的发送和接收的过程。

由于 MapReduce 框架本身设计的缺陷，在大型集群上运行表现出众多不足，如 JobTracker 是集群事务的集中处理点，存在单点故障；JobTracker 既要维护 Job 的状态又要维护 Job 的 Task 状态，造成过多的资源消耗等。

5.3.2　Hadoop 2.0 生态系统中 MapReduce 框架

为了解决 Hadoop 1.0 生态系统中 MapReduce 在可扩展性、内存消耗、线程模型、可靠性和性能上的缺陷。从 Hadoop 0.23.0 版本开始，Hadoop 的 MapReduce 框架完全重构，即 Hadoop 2.0 生态系统的 MapReduce 框架(MRv2)。第二代 MapReduce(MRv2)与第一代 MapReduce(MRv1)在编程接口、数据处理引擎方面 Map 和 Reduce 是完全一样的，即 MRv2 重用了 MRv1 的这些模块。它们之间的不同之处在于：资源管理和作业管理。MRv1 中资源管理和作业管理均是由 JobTracker 实现的，集两个功能于一体；

MRv2 中将这两部分分开了，其中，作业管理由 ApplicationMaster 实现，而资源管理由新增系统 YARN 完成。由于 YARN 具有通用性，因此基于 YARN 的 MapReduce 框架可称为“MapReduce on YARN”，该框架如图 5.3 所示。

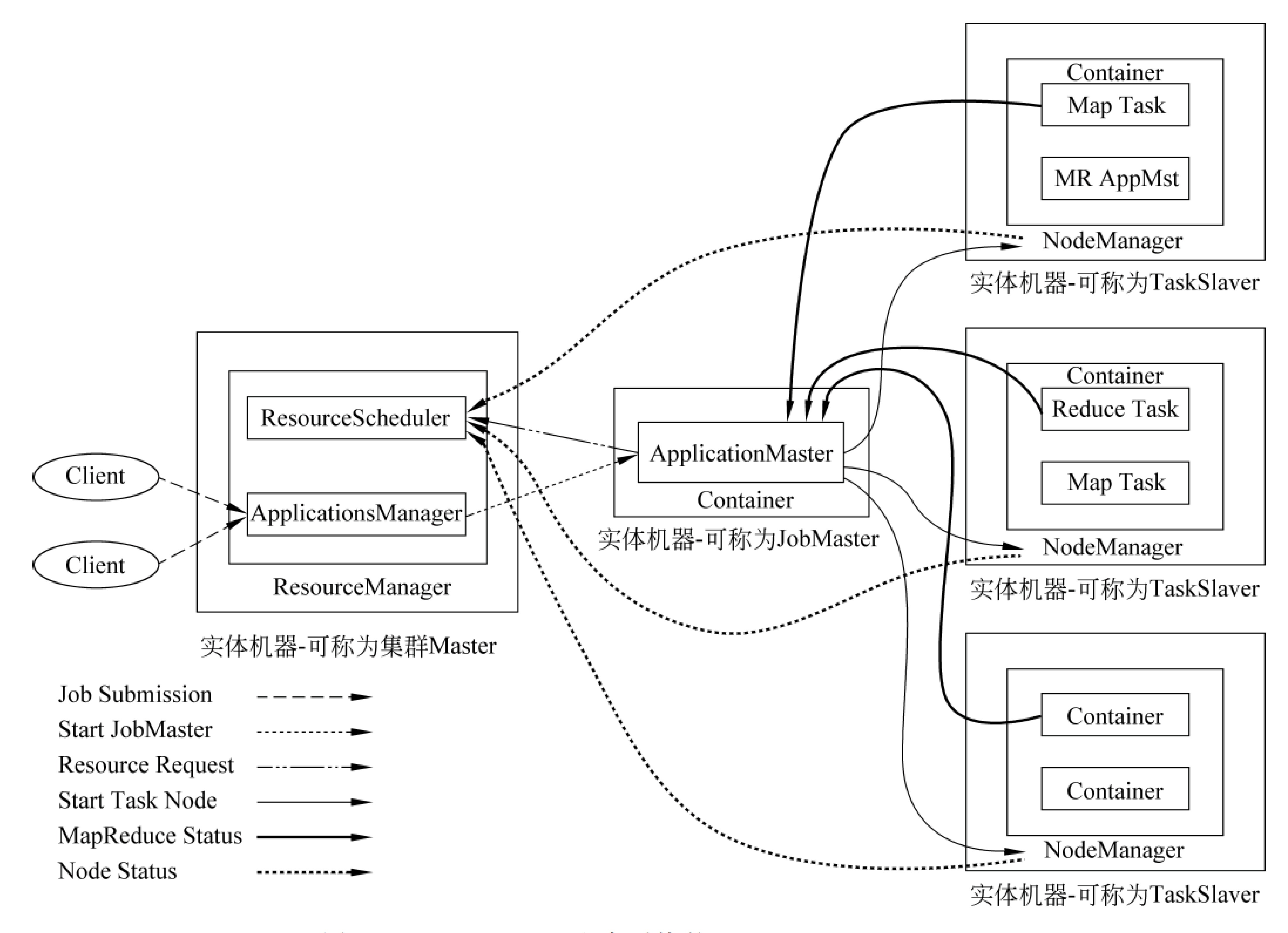


图 5.3　Hadoop 2.0 生态系统的 MapReduce on YARN

从图 5.3 中可以看出，ResourceManager 是该框架的核心，用于调度、启动每个 Job 所属的 ApplicationMaster，以及监控 ApplicationMaster 的存在情况。而且，MapReduce (MRv2)将 MRv1 中的 JobTracker 的功能分离成单独的组件，即 ApplicationMaster 和 ResourceManager。其中，ApplicationMaster 负责作业的调度和协调；ResourceManager 负责对内存、CPU、磁盘、网络等资源进行管理。NodeManager 相当于 MRv1 中的 TaskTracker，监控应用程序的资源使用情况并向 ResourceManager 汇报。

新的 MapReduce(MRv2)框架与旧的 MapReduce(MRv1)框架相比，原框架中核心的 JobTracker 和 TaskTracker 被 ResourceManager、ApplicationMaster 和 NodeManager 所取代。这种设计大大减小了原框架中 JobTracker 的资源消耗，而且 ApplicationMaster 是一个可变部分，用户可以对不同的编程模型编写自己的 AppMst，大大提高了原框架的可扩展性。另外，新旧框架的配置文件、启停脚本及全局变量等也发生了变化，读者在使用时需要根据不同的 MapReduce 框架进行相应的配置。当用户在使用 MapReduce 框架时，由于 MapReduce 框架对开发使用者透明化，其调用 API 及接口大部分都保持兼容，因此对用户的影响并不大。

5.4　MapReduce 开发环境

MapReduce 是一个能够对大量数据进行分布式处理的并行计算框架，但在编写、调试 MapReduce 程序时会带来很大难度。针对此问题，Hadoop 开发者提供了 Hadoop Eclipse 插件，从而方便开发者在 Eclipse 图形界面中编写、调试和运行 MapReduce 程序。

5.4.1　搭建 MapReduce 开发环境

MapReduce 开发环境的具体搭建过程如下。

1. 下载 Eclipse 及 Hadoop 插件

Eclipse 的下载可到官网(http://www.eclipse.org/downloads/)根据自己操作系统的版本选择 32 位或 64 位版本(本实例是在 Master1.Hadoop 节点上搭建 Hadoop 开发环境，选择了 Eclipse Linux 64 位版本)，下载完成后直接解压就可以使用 Eclipse 了。Hadoop 插件要根据 Hadoop 版本进行选择，也可以直接下载 Hadoop 源码并编译生成插件(本实例使用的 Hadoop 2.6 版本，在本书配套资料中的 softwares 目录下的 hadoop-eclipse-plugin-2.6.0.jar 为 Hadoop 2.6 的插件)，再将下载的 hadoop-eclipse-plugin-2.6.0.jar 文件放到 Eclipse 的 plugins 目录下，重启 Eclipse 即可看到该插件已生效，如图 5.4 所示。

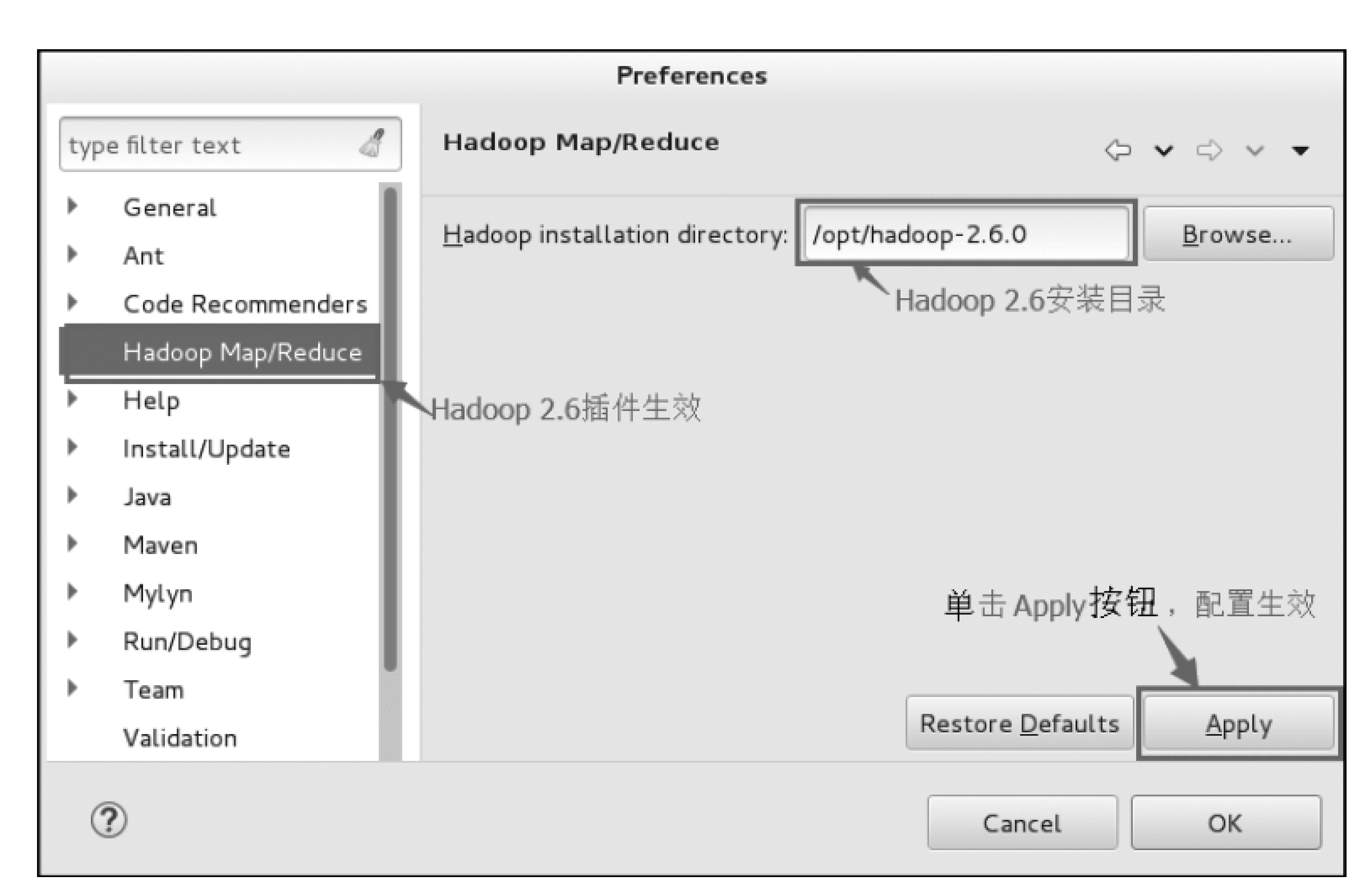


图 5.4　配置 Hadoop Map/Reduce

2. 配置 Eclipse 环境

如果 Hadoop 插件安装成功，启动 Eclipse 后，打开 Window→Preferences 选项卡，会发现 Hadoop Map/Reduce 选项，在这个选项里需要配置 Hadoop 的安装目录(本实例

Hadoop 2.6 的安装目录为/opt/hadoop-2.6.0)，配置完成后单击 Apply 按钮，如图 5.4 所示。

3. 切换 MapReduce 工作目录

切换到 MapReduce 工作目录有多种方式，可以打开 Window→Open Perspective→Other...选项卡，从中选择 Map/Reduce 选项，或者在 Eclipse 软件的右上角，单击 Java 中的按钮，从弹出的 Open Perspective 选项卡中选择 Map/Reduce 选项，然后单击 OK 按钮，即可切换到 Map/Reduce 工作目录，如图 5.5 所示。

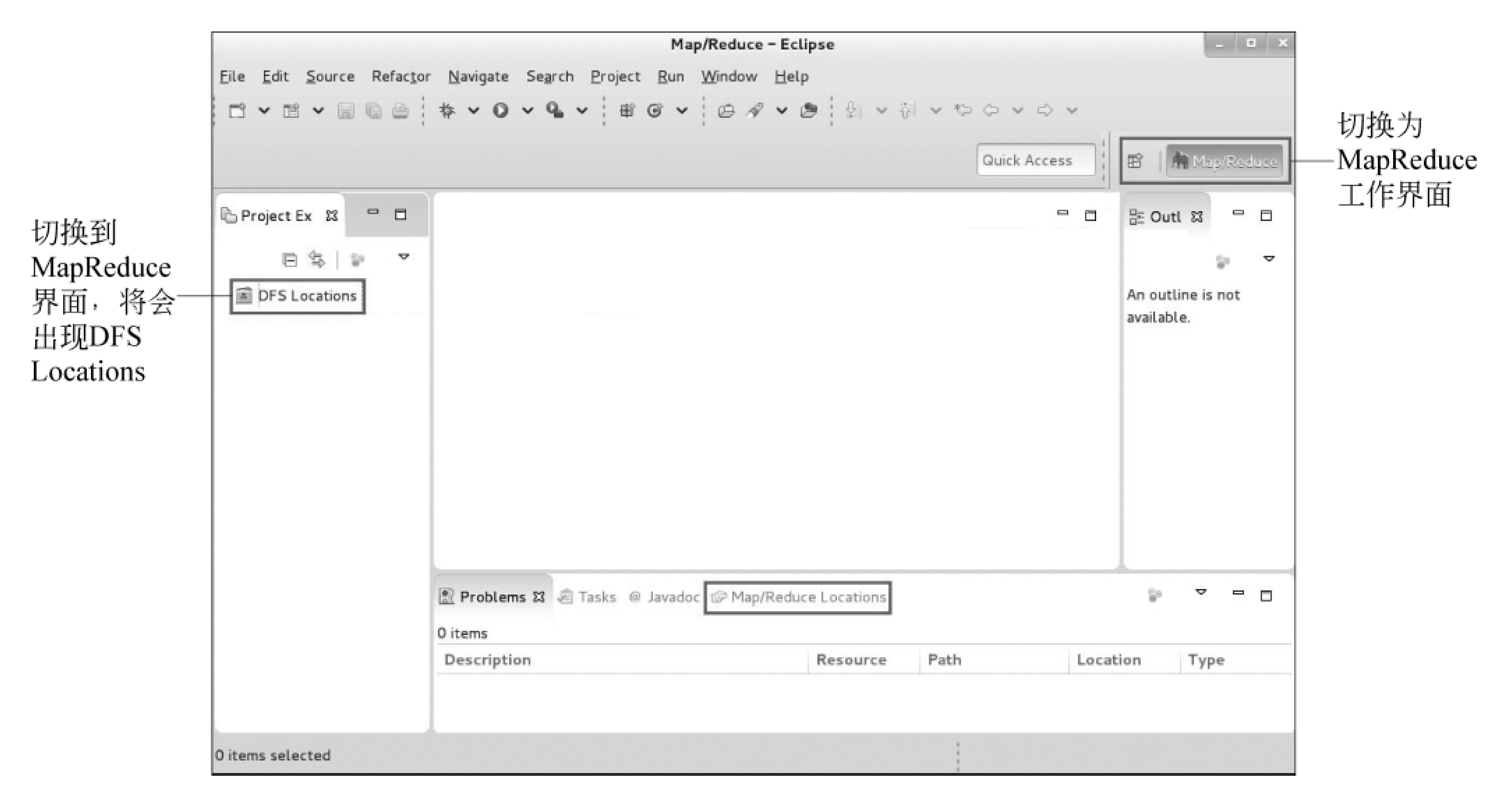


图 5.5 MapReduce 工作界面

4. 建立与 Hadoop 集群连接

单击 MapReduce 工作界面中的 MapReduce Location，在该区域右击，选择 New Hadoop Location...，如图 5.6 所示。

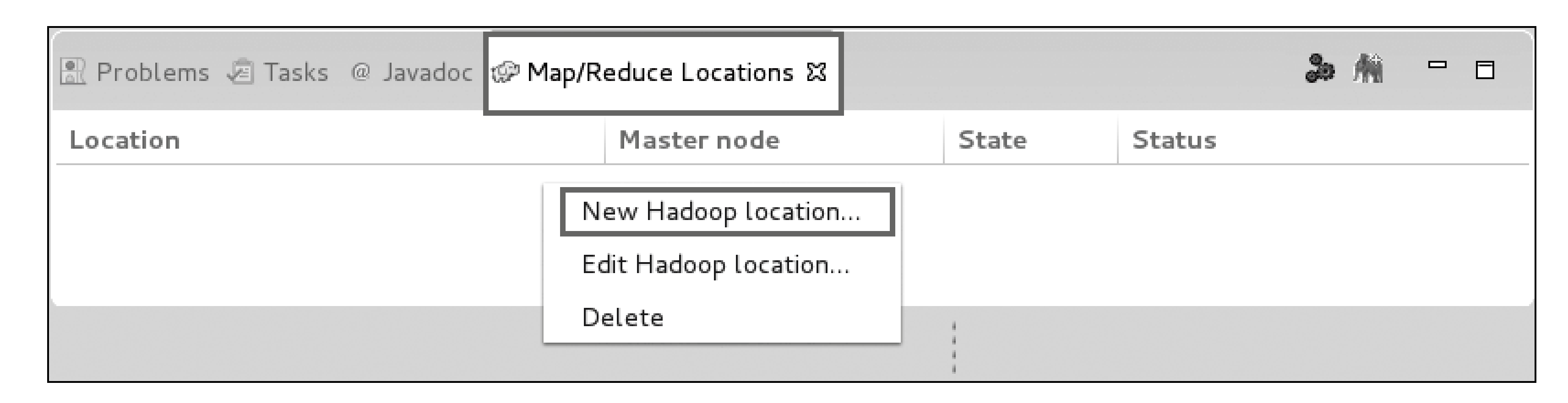


图 5.6 新建 Hadoop Location

进入 MapReduce 配置环境界面后，Location name 是标识一个 MapReduce Location(本实例定义为 MyHadoop)，Map/Reduce(V2) Master 中 Host 为 Resource Manager 机器名或 IP 地址(本实例 Rescoure Manager 为 Master1. Hadoop)，Port 为 Resource



Manager 接受任务的端口号，即 yarn-site. xml 文件中 yarn. resourcemanager. scheduler. address 配置项中的端口号（本实例在 yarn-site. xml 中配置的端口号为 8030）。DFS Master 中的 Host 为 NameNode 机器名或 IP 地址（本实例的 NameNode 为 Master1. Hadoop），Port 为 core-site. xml 文件中 fs. defaultFS 配置项中端口号（本实例在 core-site. xml 中配置的端口号为 9000）。最后，单击 Finish 按钮完成配置，如图 5.7 所示。

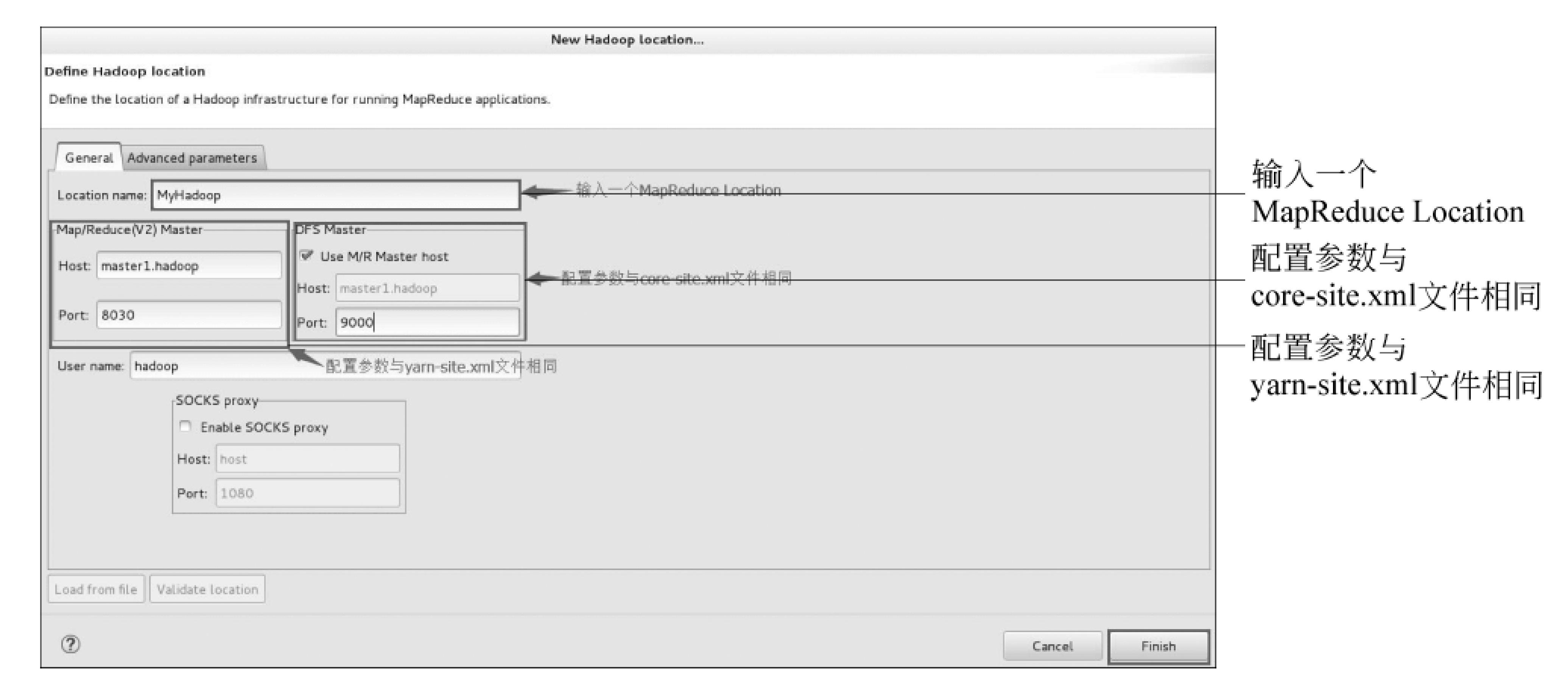


图 5.7　MapReduce 配置环境界面

单击 Finish 按钮之后，会发现在 Map/Reduce Locations 区域出现了一条刚才建立的“MapReduce Location”。如果未提示连接错误信息，则表明连接 Hadoop 集群成功，可在 Eclipse 左侧区域的 DFS Locations 中出现 CentOS HDFS 的目录树，该目录为 HDFS 文件系统中的目录结构和文件（本实例 HDFS 文件系统为空，因此显示为 0），并且右击空白区域选择相应的操作可对 HDFS 进行操作，如图 5.8 所示。如果提示连接错误信息，请检查 Hadoop 是否启动，以及 Eclipse 配置是否正确。

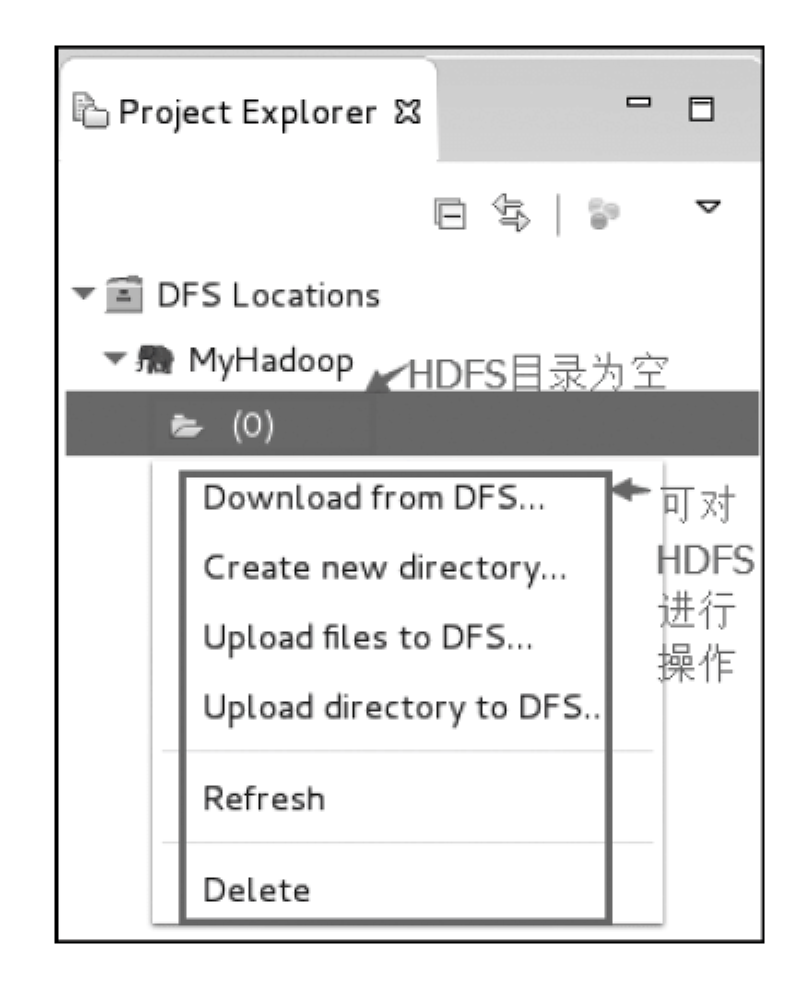


图 5.8　HDFS 目录结构

到此为止，Hadoop 的 Eclipse 开发环境已经配置完毕，读者可以上传本地文件到集群的 HDFS 分布式文件上进行验证。

5.4.2 开发 MapReduce 应用程序

MapReduce 的 Eclipse 开发环境配置完成后，在 Eclipse 图形界面中编写、调试和运行 MapReduce 程序将更加便捷。下面将详细介绍如何在 Eclipse 中开发 MapReduce 应用程序（以 Hadoop 自带的 WordCount 项目为例，请查看本书配套资料中 HadoopWorksapces 文件夹下的 WordCount 项目），具体操作步骤如下。

1. 上传测试数据

在保证 Hadoop 集群已启动的前提下，通过 Eclipse 左侧 DFS Locations 的 CentOS HDFS 目录树来完成测试数据的上传。在本实例中，在 HDFS 上创建了一个名为“input”的文件夹（作为文件输入目录），在该文件夹下导入了本地文件名为“wordcount”的文本，该文本内容为：

```
Hello World!
Hadoop!
```

文件上传过程如图 5.9 所示。

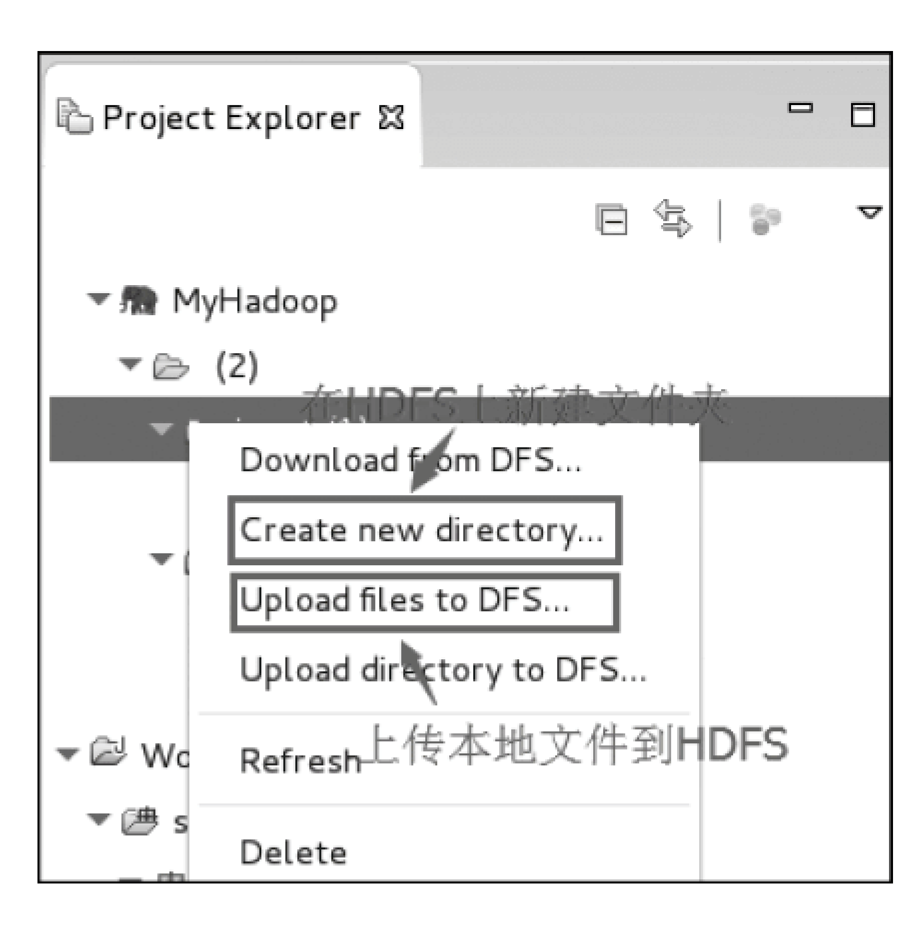


图 5.9 上传本地测试数据到 HDFS

文件上传成功后，将在 Eclipse 左侧 DFS Locations 的 CentOS HDFS 目录树中出现刚刚创建的文件内容，或者在 HDFS 文件系统中查看（浏览器地址栏中输入“http://192.168.85.100:50070/”），如图 5.10 所示。

2. 创建 WordCount 项目

在 Eclipse 的菜单栏中选择 File→New→Other...选项卡，从中选择 Map/Reduce Project（如果没有该选项，请检查 Hadoop Eclipse 插件是否安装），如图 5.11 所示。



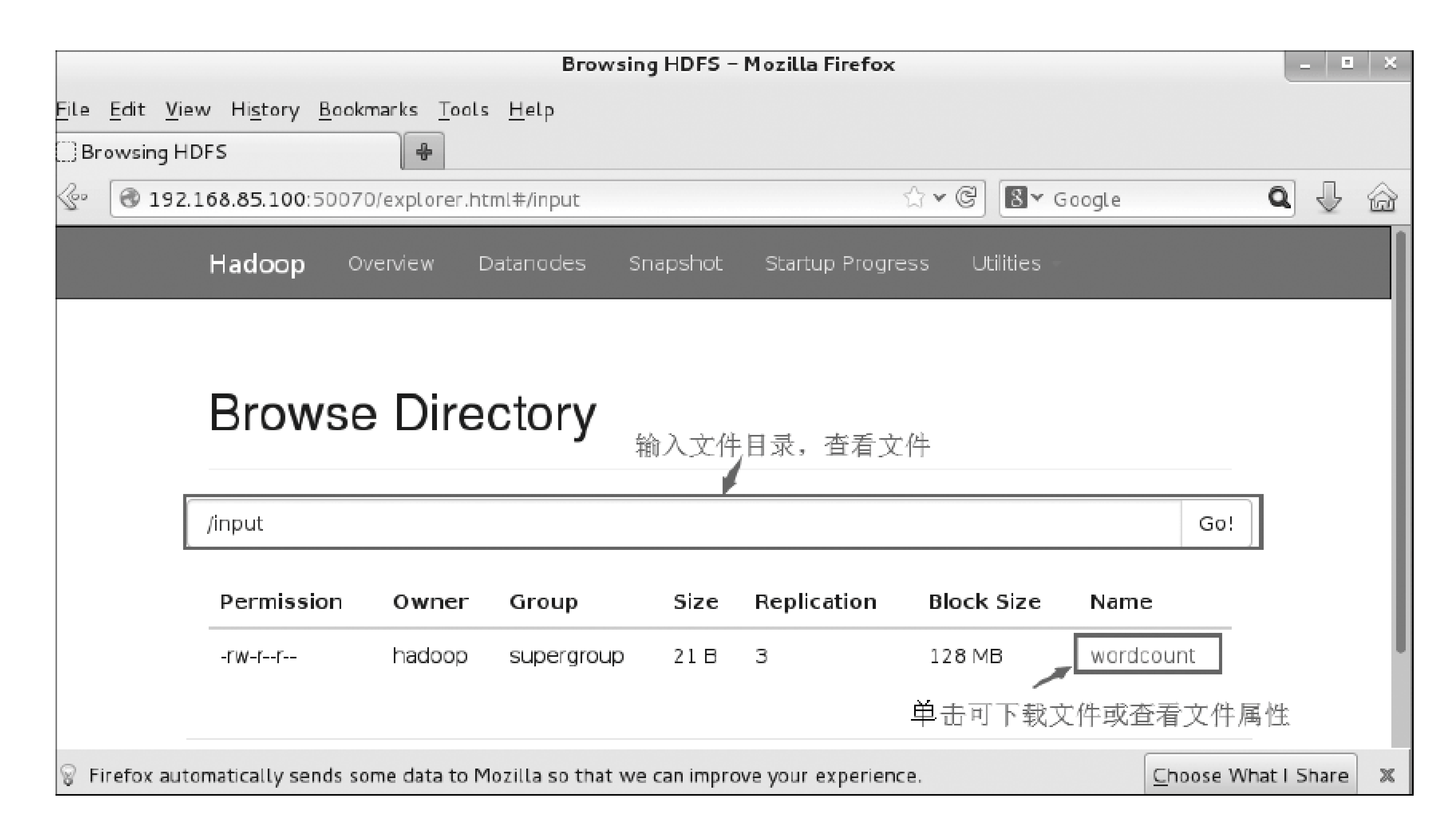


图 5.10　本地文件成功上传到 HDFS

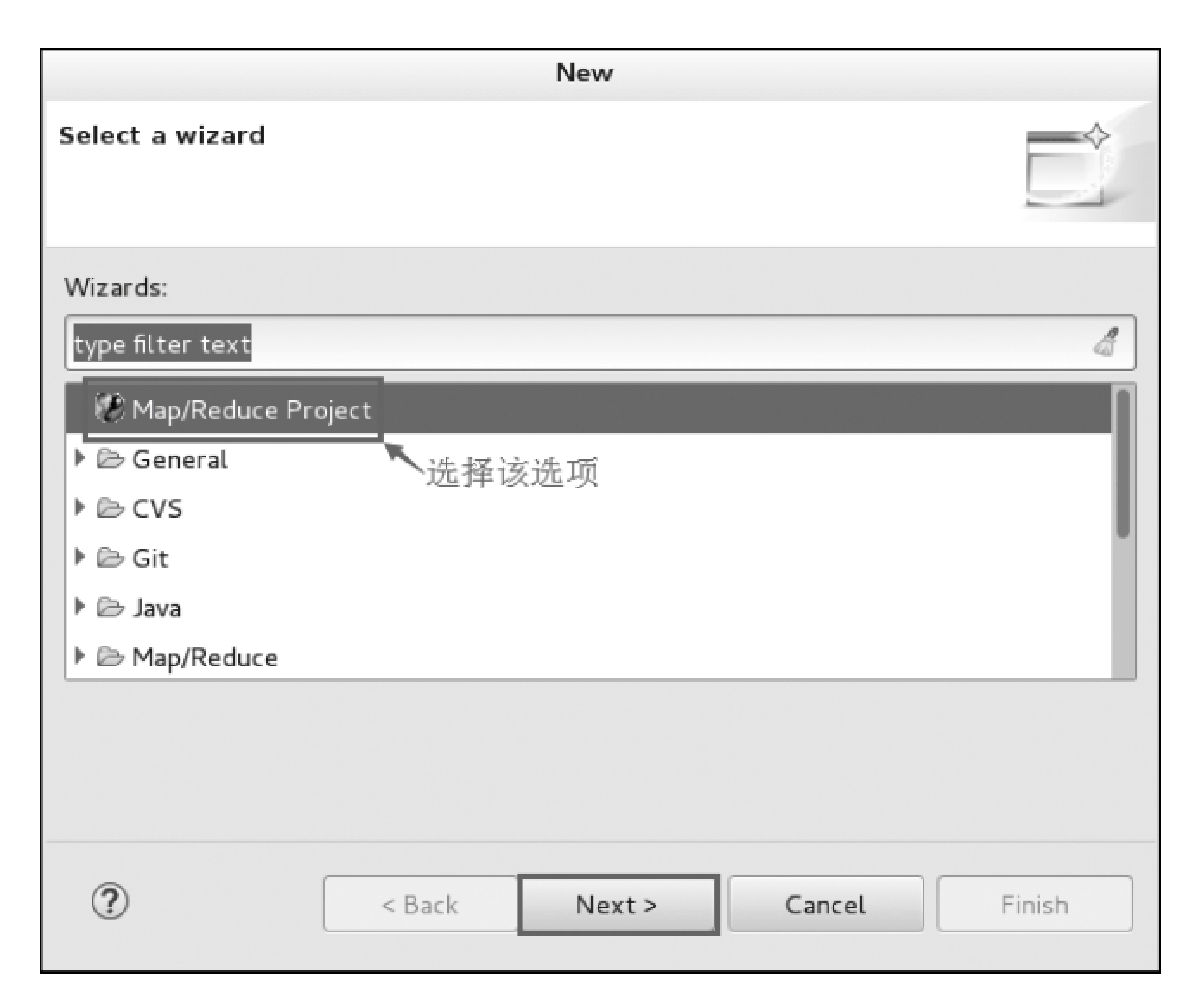


图 5.11　创建 MapReduce 项目选项卡

单击 Next 按钮进入 MapReduce 项目设置选项卡，其中项目名为“WordCount”，再选择 Use Default Hadoop，并设置 Hadoop 的安装路径。本实例 Hadoop 的安装路径为 /opt/hadoop-2.6.0。最后，单击 Finish 按钮完成 MapReduce 项目设置，如图 5.12 所示。

3. 创建 WordCount 类

在 WordCount 项目中，创建一个名为“WordCount”的类，在这里直接用 Hadoop 2.6 自带的 WordCount 程序，所以类名需要和代码中的一致为“WordCount”，包名也必须一致为“org.apache.hadoop.mapred”。最后，单击 Finish 按钮完成 WordCount 类的创建，如图 5.13 所示。

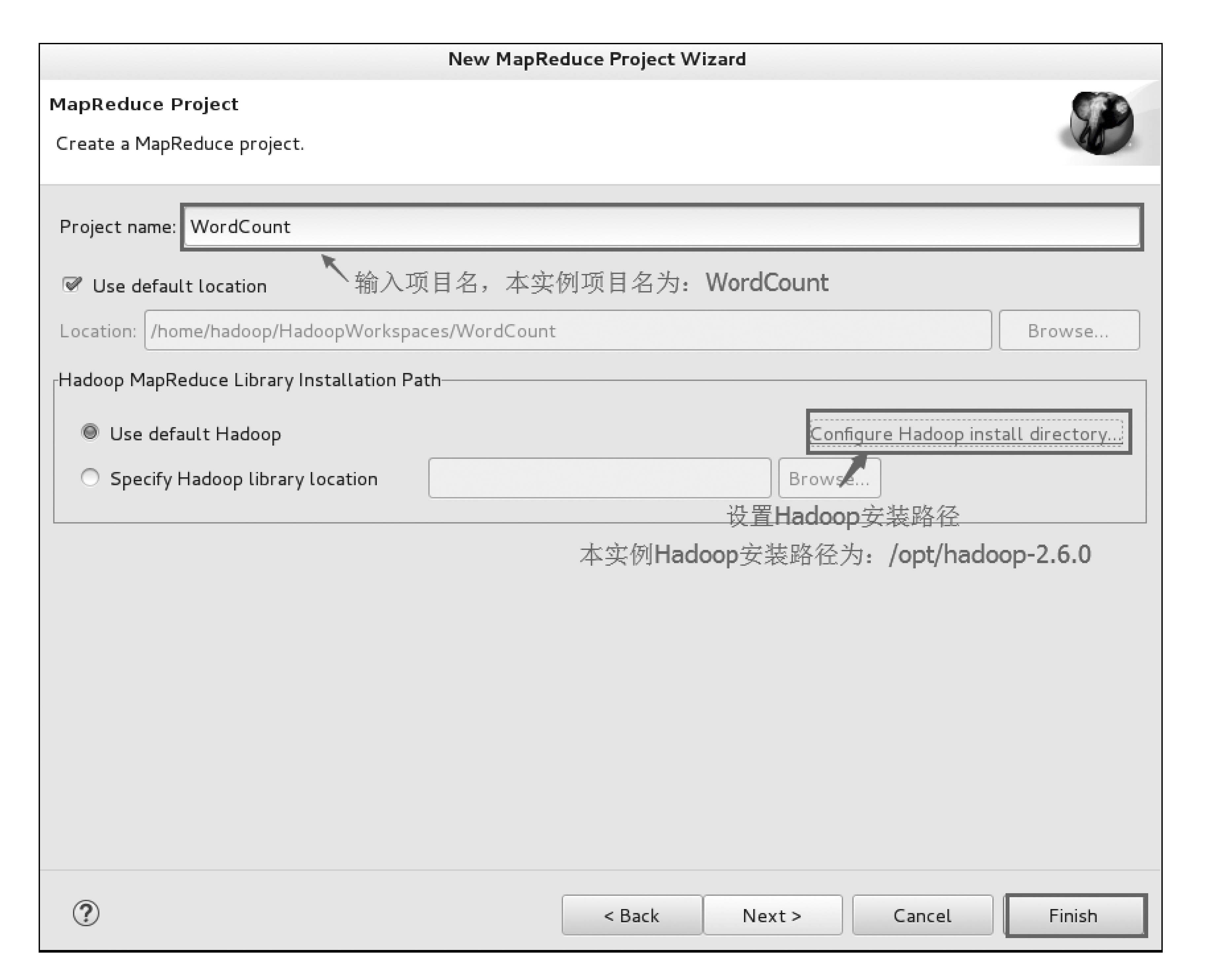


图 5.12　MapReduce 项目设置选项卡

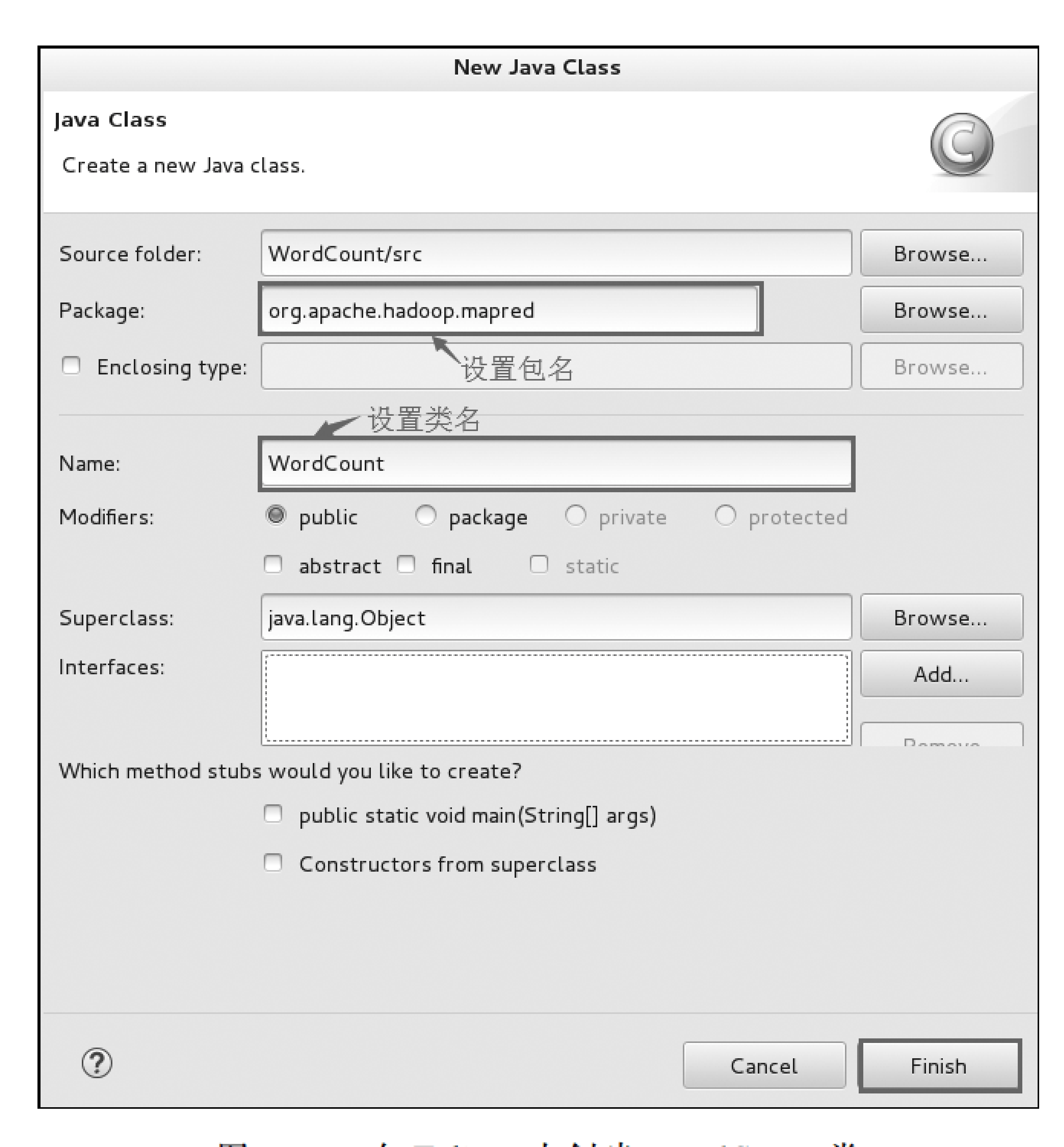


图 5.13　在 Eclipse 中创建 WordCount 类

直接用 Hadoop 2.6 自带的 WordCount 程序(源码在\hadoop-2.6.0-src\hadoop-2.6.0-src\hadoop-mapreduce-project\hadoop-mapreduce-examples\目录下),WordCount 类的代码内容如下。

```
package org.apache.hadoop.mapred;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.StringTokenizer;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

public class WordCount extends Configured implements Tool {
  public static class MapClass extends MapReduceBase
    implements Mapper<LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one=new IntWritable(1);
    private Text word=new Text();
    public void map(LongWritable key, Text value,
                    OutputCollector<Text, IntWritable> output,
                    Reporter reporter) throws IOException {
      String line=value.toString();
      StringTokenizer itr=new StringTokenizer(line);
      while(itr.hasMoreTokens()){
        word.set(itr.nextToken());
        output.collect(word, one);
      }
```

```
        }
    }
    public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator<IntWritable> values,
                       OutputCollector<Text, IntWritable> output,
                       Reporter reporter) throws IOException {
        int sum=0;
        while(values.hasNext()){
            sum+=values.next().get();
        }
        output.collect(key, new IntWritable(sum));
    }
}
static int printUsage(){
    System.out.println("wordcount [-m <maps>] [-r <reduces>] <input><output>");
    ToolRunner.printGenericCommandUsage(System.out);
    return -1;
}
public int run(String[] args)throws Exception {
    JobConf conf=new JobConf(getConf(), WordCount.class);
    conf.setJobName("wordcount");
    conf.setOutputKeyClass(Text.class);
    conf.setOutputValueClass(IntWritable.class);
    conf.setMapperClass(MapClass.class);
    conf.setCombinerClass(Reduce.class);
    conf.setReducerClass(Reduce.class);
    List<String> other_args=new ArrayList<String>();
    for(int i=0; i<args.length;++i){
        try {
            if("-m".equals(args[i])){
                conf.setNumMapTasks(Integer.parseInt(args[++i]));
            } else if("-r".equals(args[i])){
                conf.setNumReduceTasks(Integer.parseInt(args[++i]));
            } else {
                other_args.add(args[i]);
            }
        } catch(NumberFormatException except){
            System.out.println("ERROR: Integer expected instead of "+args[i]);
            return printUsage();
        } catch(ArrayIndexOutOfBoundsException except){
            System.out.println("ERROR: Required parameter missing from "+
                                       args[i-1]);
```

```
            return printUsage();
          }
        }
        if(other_args.size()!=2){
          System.out.println("ERROR: Wrong number of parameters: "+
                                    other_args.size()+" instead of 2.");
          return printUsage();
        }
        FileInputFormat.setInputPaths(conf, other_args.get(0));
        FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
        JobClient.runJob(conf);
        return 0;
      }
      public static void main(String[] args)throws Exception {
        int res=ToolRunner.run(new Configuration(), new WordCount(), args);
        System.exit(res);
      }
    }
```

4. 运行 WordCount 程序

运行 WordCount 程序前，需要配置运行参数。右击 WordCount 项目，选择 Run→Run Configurations...命令设置运行参数，需要在 Arguments 选项卡中填写 WordCount 运行的输入路径和输出路径参数。本实例的输入路径为"hdfs://master1.hadoop:9000/input/"；输出路径为"hdfs://master1.hadoop:9000/output/"，如图 5.14 所示。

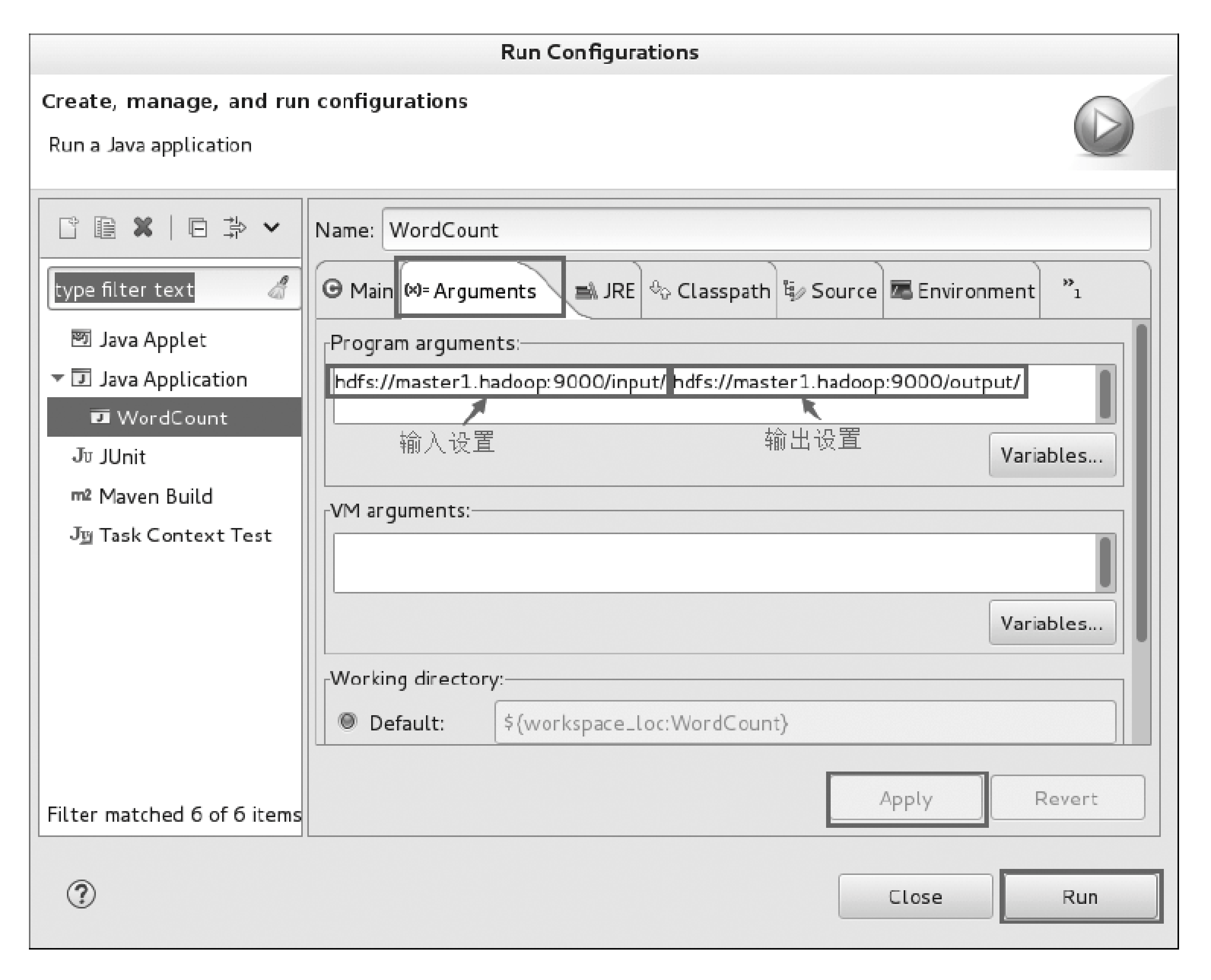


图 5.14　配置 WordCount 运行参数

配置完 WordCount 运行参数后，单击 Apply→Run 按钮就完成了 WordCount 程序的运行（首次运行时，需要选择 Run on Hadoop）。

5. 查看运行结果

运行成功后，刷新 CentOS HDFS 中的输出路径，会发现多了 output 文件夹，在该文件夹下有 _SUCCESS (0.0 b, r3) 和 part-00000 (27.0 b, r3) 两个文件，如图 5.15 所示。

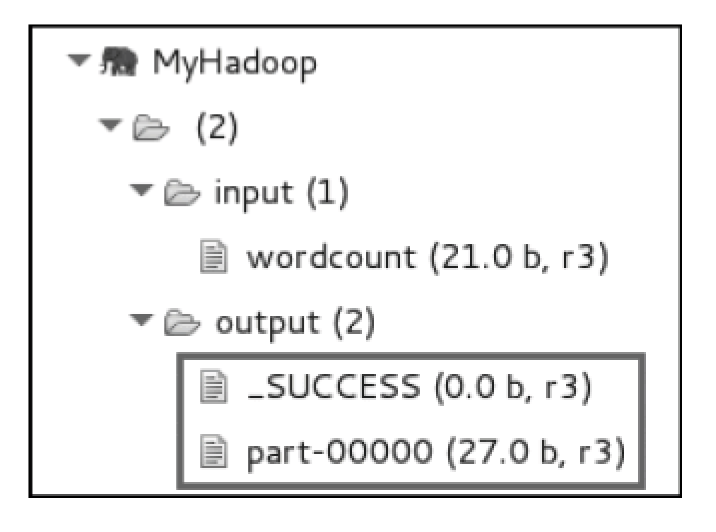


图 5.15　WordCount 运行结果

双击 part-00000 (27.0 b, r3) 文件，查看运行结果。上传的测试文件 wordcount 中只有"Hello World! Hadoop!"这几个单词，因此输出结果如下。

```
Hadoop!    1
Hello      1
World!     1
```

到此为止，一个完整的 MapReduce 应用程序开发完毕。请读者认真理解和体会，要学会触类旁通，后面章节中的项目创建、代码编写、调试和运行 MapReduce 程序，都将在 Eclipse 工具中进行。

5.5　MapReduce 编程过程

Hadoop 2.0 生态系统中 YARN 下的 MapReduce（MRv2）和 Hadoop 1.0 生态系统中的 MapReduce（MRv1）都是基于 Goolge 的 MapReduce 思想实现的。虽然两者在框架上有所区别，但是两者的调用 API 及接口大部分都保持兼容，因此从编程实现方式上保持一致。MapReduce 的编程过程如图 5.16 所示。

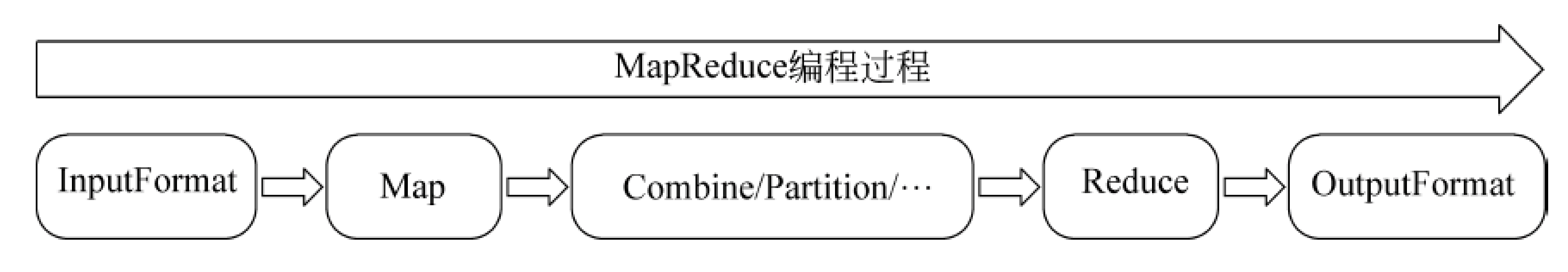


图 5.16　MapReduce 编程过程

注：在实际 MapReduce 编程过程中，只需要实现 Map 和 Reduce 即可，其他过程都已默认指定了。

5.5.1　InputFormat

在编写 MapReduce 应用程序时，首先要通过 InputFormat 设置应用程序的数据输入格式，Hadoop MapReduce 框架提供了大量的输入格式可供选择，主要区别在于文件输入格式和二进制输入格式，如表 5.1 所示为 MapReduce 几种比较常用的输入格式。

表 5.1　MapReduce 几种常用的输入格式

输入分类	输 入 格 式	说　　明
文本输入	TextInputFormat	以文本文件中的一行作为记录来进行处理，其中文本输入的格式的 Key 为 LongWriteable，Value 为 Text 类型
	KeyValueTextInputFormat	用于有行号和内容的文本文件，其中行号和文件内容是由分隔符隔开的
	NLineInputFormat	用于对少量文件数据做一些分散的并行处理任务，然后产生汇总输出
	StreamInputFormat	用于处理大型的 XML 文档
二进制输入	SequenceFileInputFormat	用于处理二进制键值对的序列，其中 Key 的类型为 IntWriteable，Value 类型为 Text
	SequenceFileAsTextInputFormat	用于顺序文件作为流操作的输入，并将 Key 和 Value 都转换为 Text 对象
	SequenceFileAsBinaryInputFormat	用于处理任意二进制的数据类型，并将顺序文件的 Key 和 Value 作为二进制对象
多样式输入	MultipleInputs	用于解决数据多样性的问题，可在每个文件上设置 InputFormat 类型
数据库输入	DBInputFormat	是从关系型数据库中读取数据的一种格式，可将数据传到集群中进行处理

要设置 MapReduce 应用程序的数据输入格式，首先要定义一个 JobConf 类对象，然后通过该对象调用 setInputFormat 设置输入数据的格式，最后设置输入目录的路径，其代码如下所示。

```
//定义一个 JobConf 类对象
JobConf conf;
//设置输入数据格式
conf.setInputFormatClass(KeyValueTextInputFormat.class);
//设置输入目录路径
FileInputFormat.setInputPaths(conf,MapReduceConfig.getInputDirectionary());
```

其中，KeyValueTextInputFormat 为我们设定的数据读取格式；FileInputFormat 是所有以文件作为数据源的 InputFormat 的实现。通过这条语句保证了输入文件会按照预设的格式被读取。InputFormat 接口定义了两个抽象方法：getSplits 和 createRecordReader 方法。其中，getSplits 方法根据作业的配置，将输入文件切片并返回

一个 InputSplit 类型的数组；createRecordReader 方法为 InputSplit 实例对象生成一个 RecordReader 对象。InputFormat 类的代码内容如下。

```
public abstract class InputFormat<K, V> {
  public abstract
    List<InputSplit> getSplits(JobContext context) throws IOException, InterruptedException;

  public abstract
    RecordReader<K,V> createRecordReader(InputSplit split,
                                         TaskAttemptContext context
                                        ) throws IOException,
                                                 InterruptedException;
}
```

InputSplit 类又定义了三个方法：getLength、getLocationInfo 和 getLocations 方法。其中，getLength 方法用于获取切片的大小，方便下一步根据输入切片的大小进行排序；getLocations 方法用于获取输入数据所在节点的名称；getLocationInfo 方法用于获取切片的存储信息。InputSplit 类代码内容如下。

```
public abstract class InputSplit {
  /**
   * Get the size of the split, so that the input splits can be sorted by size.
   * /
  public abstract long getLength() throws IOException, InterruptedException;

  /**
   * Get the list of nodes by name where the data for the split would be local.
   * The locations do not need to be serialized.
   * /
  public abstract
    String[] getLocations() throws IOException, InterruptedException;

  /**
   * Gets info about which nodes the input split is stored on and how it is
   * stored at each location.
   *
   * /
  @Evolving
  public SplitLocationInfo[] getLocationInfo() throws IOException {
    return null;
  }
}
```

RecordReader 类实现了 Closeable 接口，用于将输入数据切片转换成 key/value 对，

作为 Map 的输入。该类共定义了 6 个抽象方法：initialize、nextKeyValue、getCurrentKey、getCurrentValue、getProgress 和 close 方法。其中，initialize 方法用于调用 RecordReader 对象时进行实例化；nextKeyValue 方法用于读取下一个 key/value 对，读取成功后返回 true；getCurrentKey 用于获取当前 key 值，并返回当前 key 值，如果没有 key 值，则返回 null；getCurrentValue 方法用于获取当前的 value 值；getProgress 方法用于获取当前 RecordReader 处理数据的进度，并返回相应的进度值；close 方法用于关闭 RecordReader。RecordReader 类的代码内容如下。

```
public abstract class RecordReader<KEYIN, VALUEIN> implements Closeable {

  /**
   * Called once at initialization.
   * /
  public abstract void initialize(InputSplit split,
                                  TaskAttemptContext context
                                  )throws IOException, InterruptedException;

  /**
   * Read the next key, value pair.
   * /
  public abstract
  boolean nextKeyValue()throws IOException, InterruptedException;

  /**
   * Get the current key
   * /
  public abstract
  KEYIN getCurrentKey()throws IOException, InterruptedException;

  /**
   * Get the current value.
   * /
  public abstract
  VALUEIN getCurrentValue()throws IOException, InterruptedException;

  /**
   * The current progress of the record reader through its data.
   * /
  public abstract float getProgress()throws IOException, InterruptedException;

  /**
```

```
     * Close the record reader.
     * /
    public abstract void close() throws IOException;
  }
```

5.5.2 Map

当完成数据读操作之后，就要将分片后的数据作为 Map 的输入，进行 Map 阶段。例如，当数据被分片为 N 时，默认会给 N 个 Map 来进行处理。Map 阶段需要实现 Mapper 接口，同时继承 MapReduceBase，最后再编写 map 方法，其原型如下。

```
  public class Map extends MapReduceBase
      implements Mapper<LongWritable, Text, Text, IntWritable> {

      public void map(LongWritable key, Text value,
                      OutputCollector<Text, IntWritable> output,
                      Reporter reporter) throws IOException {
            …                                         //编写方法
      }
    }
```

Map 类继承了 MapReduceBase 类，并实现了 Mapper 接口，该接口是一个规范类型，有 4 种形式的参数＜KEYIN，VALUEIN，KEYOUT，VALUEOUT＞，分别用来指定 map 的输入 key 值类型、输入 value 值类型、输出 key 类型和输出 value 值类型。Mapper 类的代码内容如下。

```
  public class Mapper<KEYIN, VALUEIN, KEYOUT, VALUEOUT> {

    public abstract class Context
      implements MapContext<KEYIN,VALUEIN,KEYOUT,VALUEOUT> {
    }

    /**
     * Called once at the beginning of the task.
     * /
    protected void setup(Context context) throws IOException,
    InterruptedException {
      //NOTHING
    }

    /**
     * Called once for each key/value pair in the input split. Most applications
```

```
     * should override this, but the default is the identity function.
     */
    @SuppressWarnings("unchecked")
    protected void map (KEYIN key, VALUEIN value, Context context) throws IOException,
    InterruptedException {
      context.write((KEYOUT)key, (VALUEOUT)value);
    }

    /**
     * Called once at the end of the task.
     */
    protected void cleanup(Context context)throws IOException,
    InterruptedException {
      //NOTHING
    }

    /**
     * Expert users can override this method for more complete control over the
     */
    public void run(Context context)throws IOException, InterruptedException {
      setup(context);
      try {
        while(context.nextKeyValue()){
          map(context.getCurrentKey(), context.getCurrentValue(), context);
        }
      } finally {
        cleanup(context);
      }
    }
  }
```

Mapper 类提供了 setup()、map()、cleanup()和 run()4 个方法。其中，setup()用于执行 map 之前的准备工作，cleanup()方法则在所有的 map 任务完成后被调用，这两个方法用于管理 Mapper 生命周期中的资源；map()方法用于对一次输入的 key/value 对进行 map 操作；run()方法用于执行 setup()→run()→cleanup()过程，即首先调用 setup()方法，然后迭代所有的 key/value 对进行 map 操作，最后调用 cleanup()方法。

在编写 Map 类实现 Mapper 接口类时，必须要实现 map()方法，该方法根据<K1,V1>生成<K2,V2>，最后通过 Context 输出。在 Hadoop MapReduce 中已经预定义了几种 Mapper 类供用户使用，如表 5.2 所示。

表 5.2 MapReduce 中预定义的 Mapper 列举

类	说　明
IdentityMapper<K,V>	将输入的 <key,value> 原封不动地输出为中间结果
InverseMapper<K,V>	将输入<key, value> map 为输出<value, key>
RegexMapper<K>	为每一个匹配的正则表达式生成一个(match,1)键值对
TokenCountMapper<K>	当输入值被标记时,生成一个(token,1)键值对
MultithreadedMapper<K,V>	多线程执行 map 方法

注: 具体 Mapper 实现请查看 org. apache. hadoop. maped. lib 和 org. apache. hadoop. mapreduce. lib. map 目录下相应的实现类。读者可根据实际需求去使用相应的 Mapper 类。

5.5.3 Combine/Partition

Combiner 主要负责将同一个 map 中相同的 key 进行合并,避免重复传输,从而减少传输中的通信开销;Partitioner 负责将 map 产生的中间结果进行划分,确保相同的 key 值到达同一个 value,Combine/Partition 过程如图 5.17 所示。

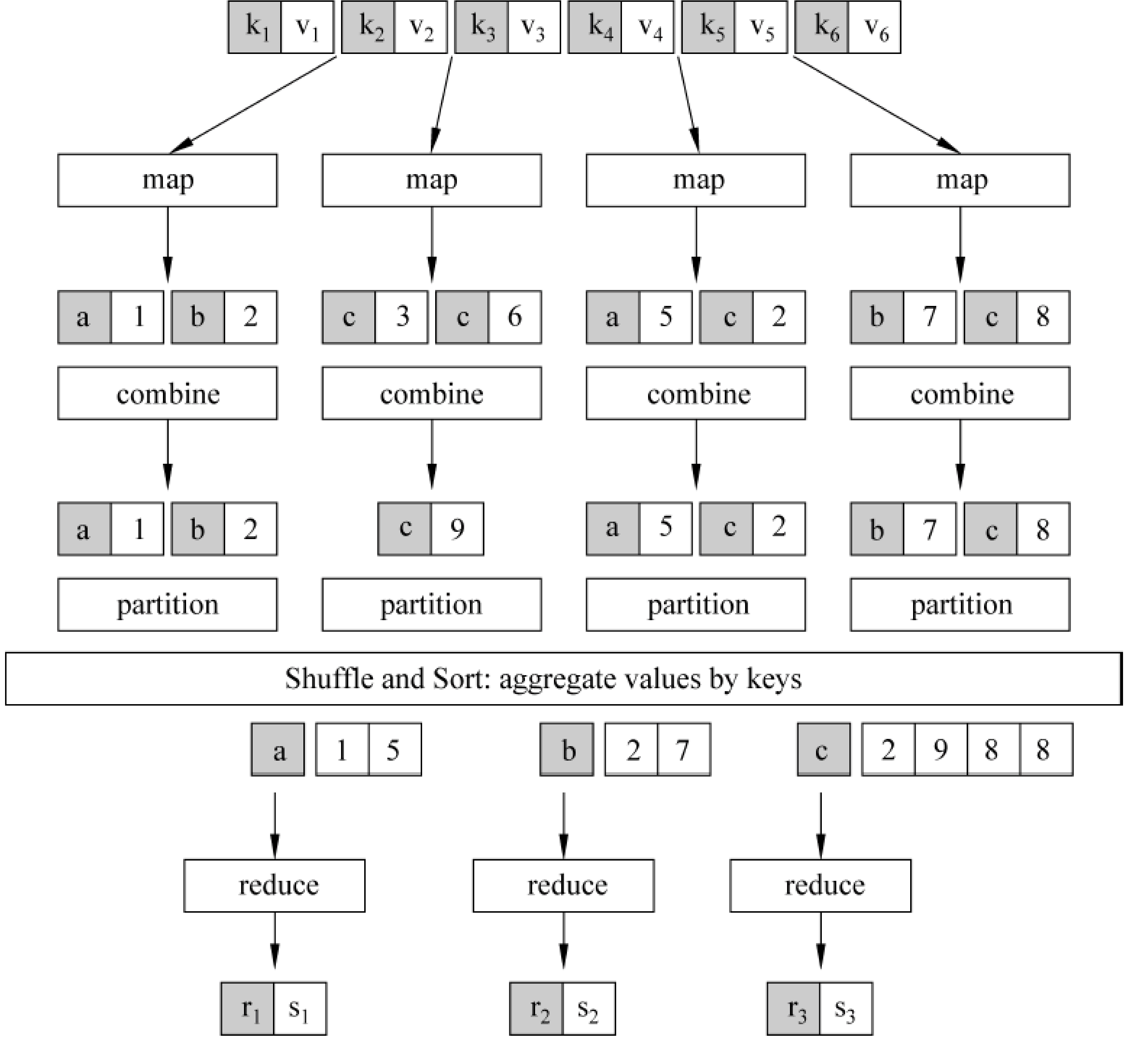


图 5.17 Combine/Partition 过程

Combine 过程其实也是一种 Reduce 操作，是 Map 运算的后续操作，在 MapReduce 下默认 Combine 过程的实现为 IdentityReducer 类，其代码内容如下。

```
public class IdentityReducer<K, V>
    extends MapReduceBase implements Reducer<K, V, K, V> {

  /** Writes all keys and values directly to output. */
  public void reduce(K key, Iterator<V> values, OutputCollector<K, V> output, Reporter reporter)
    throws IOException {
    while(values.hasNext()){
      output.collect(key, values.next());
    }
  }
}
```

如在 WordCount 项目中对文件中的单词频率做统计，在 Map 计算时如果统计到 A 单词就会记录为 1，文件中该 A 单词可能出现多次，如果不进行 Combine 操作，可能会导致 Map 输出文件冗余。因此，在 Map 计算出中间文件前对相同的 key 做一个 Combine 操作，减少了中间文件输出冗余并提高了宽带的传输效率。对于 Combine 过程，并不是所有的 MapReduce 任务都需要使用，要根据实际业务情况来决定。

Partition 过程是将 Map 的运行结果发送到相应的 Reduce 中。在设计 Partition 时，需要充分考虑到如何将作业均匀分配到不同的 Reduce 的负载均衡问题和作业分配速度的效率问题。用户可以通过继承 Partitioner 接口实现自己的 Partition 过程，提高负载均衡和作业分配效率。Partitioner 接口类的代码内容如下。

```
public interface Partitioner<K2, V2> extends JobConfigurable {

  /**
   * Get the paritition number for a given key(hence record)given the total
   * number of partitions i.e. number of reduce-tasks for the job.
   *
   * <p>Typically a hash function on a all or a subset of the key.</p>
  int getPartition(K2 key, V2 value, int numPartitions);
}
```

Partitioner 中需要实现 getPartition()方法，该方法返回<K2，V2>对应的 Reduce ID。如果用户没有提供 Partition，MapReduce 会使用默认的 HashPartitioner 类，另外还提供了 BinaryPartitioner 类、KeyFieldBasedPartitioner 类和 TotalOrderPartitioner 类。其中，HashPartitioner 类是 MapReduce 的默认 Partition，该类计算当前 Reduce ID 的方法如下。

```
(key.hashCode()& Integer.MAX_VALUE)          %numReduceTasks
```

BinaryPartitioner 类继承于 Partitioner 类，该类提供了 leftOffset 和 rightOffset，在

计算当前 Reduce ID 的方法时，仅对键值 K 的[rightOffset，leftOffset]这个区间取 hash；KeyFieldBasedPartitioner 类也是基于 hash 的 Partitioner，与 BinaryPartitioner 不同之处在于该类提供了多个区间用于计算 hash，当区间数为 0 时，则退化成 HashPartitioner；TotalOrderPartitioner 类不同于以上三个 Partitioner，对于 BinaryComparable 类型的 Key，采用字典排序法查找当前的 Key 所在的 Reduce ID，对于非 BinaryComparable 类型的 Key，则采用二分法查找当前的 Key 所在的 Reduce ID。

从图 5.17 中可以看出，MapReduce 还要经过 Shuffle 和 Sort 等一系列过程。其中，Shuffle 过程主要负责将完成的 Map 任务的中间输出结果交换到相应的 Reduce 中，并对某些 Map 任务通过阈值判断来决定写到磁盘还是直接放在内存中，从而确保 Map 和 Reduce 拥有足够的内存空间；MapReduce 采用了基于 Sort 的策略将 Key 值相同的数据聚集在一起，由于各个 Map 任务已经实现了对自己的处理结果进行局部排序，Sort 过程只需对所有的数据进行一次归并排序即可。

注：感兴趣的读者，可以查看 org. apache. hadoop. mapred. lib 目录下 HashPartitioner 类、BinaryPartitioner 类、KeyFieldBasedPartitioner 类和 TotalOrderPartitioner 类。读者可根据实际需求去使用相应的 Partition。

5.5.4 Reduce

通过 Map 阶段生成的中间文件数据还要经过 Reduce 阶段，从而产生最终结果数据并写到 HDFS 中。Reduce 阶段需要实现 Reducer 接口，同时继承 MapReduceBase，最后再编写 reduce 方法，其原型如下。

```
public class Reduce extends MapReduceBase
    implements Reducer< Text, IntWritable, Text, IntWritable> {

    public void reduce(Text key, Iterator< IntWritable> values,
                       OutputCollector< Text, IntWritable> output,
                       Reporter reporter) throws IOException {
      ...
    }
  }
```

Reduce 类以 Map 的输出作为输入，并继承了 MapReduceBase 类，实现了 Reducer 接口及重写了 reduce()方法。其中，Reducer 接口类有 4 种形式的参数<K2，V2，K3，V3>，其代码内容如下。

```
public interface Reducer< K2, V2, K3, V3> extends JobConfigurable, Closeable{

  void reduce(K2 key, Iterator< V2> values,OutputCollector< K3, V3> output, Reporter reporter)
```

```
        throws IOException;
}
```

Reducer 接收从 Mapper 传递而来的 key/value 对，然后根据 key 来排序，分组，并生成＜K2, list＜V2＞＞，最后 Reducer 根据＜K2, list＜V2＞＞生成＜K3, V3＞。在 Hadoop MapReduce 中已经预定义了几种 Reducer 类供用户使用，表 5.3 给出了 MapReduce 中预定义的几种 Reducer。

表 5.3　MapReduce 中预定义的 Reducer 列举

类	说　明
IndentityReducer＜K,V＞	将输入的 ＜key,value＞ 原封不动地输出为结果
LongSumReducer＜K＞	对长整型的 value 值求和
IntSumReducer	对整型的 value 值求和

注：具体 Mapper 实现请查看 org.apache.hadoop.maped.lib 和 org.apache.hadoop.mapreduce.lib.reduce 目录下相应的类。读者可根据实际需求去使用相应的 Reducer 类。

5.5.5　OutputFormat

MapReduce 使用 OutputFormat 类将数据输出并存入文件中，每个 Reduce 将它的输出数据直接写到自己的文件中。输出文件存在于一个共有目录当中，一般被命名为"part-nnnnn"，其中，nnnnn 是 Reduce 的分区 ID。Hadoop MapReduce 框架提供了几种数据的输出格式可供选择，如表 5.4 所示为 MapReduce 几种比较常用的输出格式。

表 5.4　MapReduce 几种常用的输出格式

输出分类	输 出 格 式	说　明
文本输出	TextOutputFormat	是以一行的形式进行文件写入，Key 和 Value 可以是任意类型，其分隔符默认为 Tab 符
二进制输出	SequenceFileOutputFormat	将输出写入到一个顺序文件中，其格式紧凑，而且数据可以被压缩
	SequenceFileAsBinaryOutputFormat	将 Key/Value 对当作二进制写入到一个顺序文件中
	MapFileOutputFormat	将排序后的 Key/Value 对写入到一个 mapfile 文件中
多个输出	MultipleTextOutputFormat	将结果输出到多个文件中，其中根据 Key/Value 对进行分区
	MultipleSequenceFileOutputFormat	将结果输出到多个顺序文件中，其中根据 Key/Value 对进行分区
延迟输出	LazyOutputFormat	延迟输出格式，可以保证在第一条记录输出的时候才真正创建文件
数据库输出	DBOutputFormat	向关系型数据库中写入数据的一种格式

要设置 MapReduce 应用程序的数据输出格式，需要通过 OutputFormat 的对象调用 setOutputFormatClass 设置输入数据的格式，最后设置输入目录的路径，其代码如下所示。

```
conf.setOutputFormatClass(SequenceFileOutputFormat.class);
FileOutputFormat.setOutputPath(conf, new Path());
```

OutputFormat 类指定输出数据的格式，该类定义了三个方法：getRecordWriter、checkOutputSpecs 和 getOutputCommitter 方法。其中，getRecordWriter 方法用于写入输出结果；checkOutputSpecs 方法用于检查结果输出的存储空间是否有效；getOutputCommitter 方法用于任务提交。OutputFormat 类的代码内容如下。

```
public abstract class OutputFormat<K, V> {

  /**
   * Get the {@link RecordWriter} for the given task.
   * /
  public abstract RecordWriter<K, V>
    getRecordWriter(TaskAttemptContext context)throws IOException,
    InterruptedException;

  /**
   * Check for validity of the output-specification for the job.
   * /
  public abstract void checkOutputSpecs(JobContext context)throws IOException, InterruptedException;

  /**
   * Get the output committer for this output format. This is
   * responsible for ensuring the output is committed correctly.
   * /
  public abstract
  OutputCommitter getOutputCommitter (TaskAttemptContext context) throws IOException,
  InterruptedException;
}
```

代码中的 RecordWriter 类主要负责根据存储环境的需要先对一个 key/value 对进行组织，然后把这个组织好的 key/value 存储到某个存储环境中，每一行中的 key 和 value 是通过分隔符 Tab 来区别的。RecordWriter 类的代码内容如下。

```
public abstract class RecordWriter<K, V> {
  /**
   * Writes a key/value pair.
```

```
 * /
public abstract void write(K key, V value)throws IOException,
InterruptedException;

/**
 * Close this <code>RecordWriter</code> to future operations.
 * /
public abstract void close(TaskAttemptContext context)throws IOException, InterruptedException;
}
```

OutputCommitter 类主要负责对任务的输出进行管理,代码内容如下。

```
public abstract class OutputCommitter {
  /**
   * Setup Job
   * /
  public abstract void setupJob(JobContext jobContext)throws IOException;

  /**
   * For cleaning up the job's output after job completion.
   * /
  @Deprecated
  public void cleanupJob(JobContext jobContext)throws IOException { }

  /**
   * For committing job's output after successful job completion.
   * /
  public void commitJob(JobContext jobContext)throws IOException {
    cleanupJob(jobContext);
  }


  /**
   * For aborting an unsuccessful job's output.
   * /
  public void abortJob(JobContext jobContext, JobStatus.State state)
  throws IOException {
    cleanupJob(jobContext);
  }

  /**
   * Sets up output for the task.
```

```
 * /
public abstract void setupTask(TaskAttemptContext taskContext)
throws IOException;

/**
 * Check whether task needs a commit.
 * /
public abstract boolean needsTaskCommit(TaskAttemptContext taskContext)
throws IOException;

/**
 * To promote the task's temporary output to final output
 * /
public abstract void commitTask(TaskAttemptContext taskContext)
throws IOException;

/**
 * Discard the task output.
 * /
public abstract void abortTask(TaskAttemptContext taskContext)
throws IOException;

/**
 * Is task output recovery supported for restarting jobs?
 * /
@ Deprecated
public boolean isRecoverySupported() {
  return false;
}

/**
 * Is task output recovery supported for restarting jobs?
 * /
public boolean isRecoverySupported(JobContext jobContext) throws IOException {
  return isRecoverySupported();
}

/**
 * Recover the task output.
 * /
public void recoverTask(TaskAttemptContext taskContext)
throws IOException
```

```
    {}
}
```

注：感兴趣的读者可以查看 OutputCommitter 类的具体实现类 FileOutputCommitter，理解应该如何根据自己的实际情况来定义对任务的输出管理。

5.6　MapReduce 开发实例

在 Eclipse 中开发 MapReduce 应用程序，便于开发者编辑和编译 MapReduce 应用程序，并对程序进行单元测试、作业调优、运行调试等。下面将重点介绍 WordCount 项目实例，以便让读者更深刻地理解如何在 Eclipse 中开发 MapReduce 应用程序。

5.6.1　MapReduce 编程

通过 MapReduce 编程实现对 HDFS 上指定文件目录中所有文本中单词出现的次数，即 WordCount 项目。因为有多个文件，可以并行地统计每个文本中单词出现的个数，最后进行合并计算，即 MapReduce 编程。MapReduce 应用程序编程的关键步骤如下。

1. 编写 WordCount 类

首先，新建一个 MapReduce 工程（创建过程请参考 5.4.2 节），并新建一个 WordCount 类。在 WordCount 类中要实现 Mapper 接口（编写 Map 类）和 Reducer 接口（编写 Reduce 类），并进行 Job 的配置，代码内容如下。

```
package org.apache.hadoop.mapred;                    //WordCount 类所在的包结构

//下面引入的几个类是 Java 中经常用到的，不是 Hadoop 特有
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.StringTokenizer;

//下面引入的类全部是 Hadoop 中特有的
import org.apache.hadoop.conf.Configuration;         //配置信息
import org.apache.hadoop.conf.Configured;            //配置信息
import org.apache.hadoop.fs.Path;                    //用于将路径转换为标准的 URL 格式
import org.apache.hadoop.io.IntWritable;             //数据类型
import org.apache.hadoop.io.LongWritable;            //数据类型
import org.apache.hadoop.io.Text;                    //数据类型
import org.apache.hadoop.mapred.FileInputFormat;
                                                     //用于文件分割及<key,value>生成
```

```
import org.apache.hadoop.mapred.FileOutputFormat;
                                              //规定了输出结果的格式
import org.apache.hadoop.mapred.JobClient;    //用于提交 Job
import org.apache.hadoop.mapred.JobConf;      //用于配置 Job
import org.apache.hadoop.mapred.MapReduceBase;
                                              //Map 和 Reduce 类都要继承的基类
import org.apache.hadoop.mapred.Mapper;       //Map 类需要实现的接口
import org.apache.hadoop.mapred.OutputCollector;
                                              //用于生成 Map/Reduce 的输出结果
import org.apache.hadoop.mapred.Reducer;      //Reduce 类需要实现的接口
import org.apache.hadoop.mapred.Reporter;     //用于向 JobTracker 报告任务执行情况
import org.apache.hadoop.util.Tool;           //Tool 接口类
import org.apache.hadoop.util.ToolRunner;     //用于调用 run()方法

public class WordCount extends Configured implements Tool {
    //实现 Mapper 接口
    //实现 Reducer 接口
    //Job 配置
  public static void main(String[] args)throws Exception {
    //MapReduce 程序运行
  }
}
```

2. 实现 Mapper 接口

在 WordCount 类中编写 Map 类，该类实现 Mapper 接口，并规定如何将输入的<key，value>对转化为中间结果的<key，list of values>对，Map 类的代码内容如下。

```
//编写 Map 静态内部类，类名为 Map，继承自 MapReduceBase，并实现 Mapper 接口
public static class Map extends MapReduceBase
    implements Mapper< LongWritable, Text, Text, IntWritable> {

    private final static IntWritable one=new IntWritable(1);
                                        //one 为 Map 输出的 value
    private Text word=new Text();       //拆分出的每个 word 作为 Map 输出的 key
    public void map(LongWritable key, Text value,
                    OutputCollector< Text, IntWritable> output,
                    Reporter reporter)throws IOException {
      String line=value.toString();     //将 value 转换为 Java 的 String 处理
      StringTokenizer itr=new StringTokenizer(line);
                                        //itr 用于拆分字符串
      while(itr.hasMoreTokens()){
        word.set(itr.nextToken());
```

```
            output.collect(word, one);                //output 收集生成的中间结果<word,one>
        }
    }
}
```

3. 实现 Reducer 接口

在 WordCount 类中编写 Reduce 类,该类实现 Reducer 接口,并规定如何对 Map 输出的中间结果<key, list of values>进一步处理,转化为最终的结果输出<key,value>对,Reduce 类的代码内容如下。

```
//编写 Reduce 内部类,类名为 Reduce,继承自 MapReduceBase,并实现 Reducer 接口
    public static class Reduce extends MapReduceBase
    implements Reducer<Text, IntWritable, Text, IntWritable>{

    public void reduce(Text key, Iterator<IntWritable> values,
                        OutputCollector<Text, IntWritable> output,
                        Reporter reporter) throws IOException {
      int sum=0;
      while(values.hasNext()){
        sum+=values.next().get();                //对 key 相同的所有 value 进行累加
      }
      output.collect(key, new IntWritable(sum));
                                                 //output 收集输出结果
    }
}
```

4. 配置作业

所有的 MapReduce 作业都需要有一个 JobConf 对象来配置实际的 MapReduce 任务和提交 MapReduce 任务。有关 JobConf 类所提供的成员参数和方法请查看 Hadoop API 中的 org. apache. hadoop. mapred. JobConf。本实例在 WordCount 类中定义一个公有的 run()方法,用于 Job 的配置和提交,代码内容如下。

```
public int run(String[] args)throws Exception {
    JobConf conf=new JobConf(getConf(), WordCount.class);     //设置 Job 所在的主类
    conf.setJobName("wordcount");                             //设置 Job 名
    //设置 Job 输出结果<key,value>中的 key 和 value 数据类型
    conf.setOutputKeyClass(Text.class);                       //key 为 Text 型
    conf.setOutputValueClass(IntWritable.class);              //value 为 IntWritable 型
    conf.setMapperClass(Map.class);                           //设置完成 Map 任务的类
    conf.setCombinerClass(Reduce.class);                      //设置完成 Combine 任务的类
```

```
        conf.setReducerClass(Reduce.class);             //设置完成 Reduce 任务的类
        //设置输入数据所在的 HDFS 的路径
        FileInputFormat.setInputPaths(conf, other_args.get(0));
        //设置输出结果保存在 HDFS 的路径
        FileOutputFormat.setOutputPath(conf, new Path(other_args.get(1)));
        JobClient.runJob(conf);                         //提交 Job
    }
```

5. 代码测试

编写完 MapReduce 应用程序后，就需要对代码进行单元测试，当 map 和 reduce 功能正确后，再在小规模集群上对 MapReduce 进行集成测试，测试成功后就可以在 Hadoop 集群上运行。有关 MapReduce 的测试，可使用 MRUnit 进行单元测试和集成测试，如用 MapDriver 单元测试 Map、用 ReduceDriver 单元测试 Reduce、用 MapReudceDriver 集成测试 MapReduce 作业、用 PiplineMapReduceDriver 多 MapReduce 组合的集成测试（有关 MRUnit 的细节请查看 http://mrunit.apache.org/）。使用 MRUint 对 Map 功能的单元测试如下所示。

```
public class WordCountMapperTest {
    private Map map;
    private MapDriver driver;

    @Before
    public void init(){
        map=new WordCountMap();
        driver=new MapDriver(map);
    }

    @Test
    public void test()throws IOException {
        String line="Hello Hadoop";
        driver.withInput(null, new Text(line))
        .withOutput(new Text("Hello"), new IntWritable(1))
        .withOutput(new Text("Hadoop"), new IntWritable(1))
        .runTest();
    }
}
```

假定 Map 输入内容为“Hello Hadoop”，通过 MapDriver 的 withInput 和 withOutput 组织 Map 的输入键值和期待的输出键值，并通过 runTest()方法运行作业，测试 map 功能。同理，测试 Reduce 功能同 Map 功能相同，其测试代码内容如下。

```
public class WordCountReducerTest {
    private Reduce reduce;
    private ReduceDriver driver;

    @Before
    public void init(){
        reduce=new WordCountReduce();
        driver=new ReduceDriver(reduce);
    }

    @Test
    public void test()throws IOException {
        String key="Hadoop";
        List values=new ArrayList();
        values.add(new IntWritable(2));
        values.add(new IntWritable(4));
        driver.withInput(new Text("Hadoop"), values)
        .withOutput(new Text("Hadoop"), new IntWritable(6))
        .runTest();
    }
}
```

这里假设 Reduce 输入为<Hadoop,6>,则期待 Reduce 的输出结果应该为<Hadoop,6>。如果结果一致,则测试运行通过;如果结果不一致,请重新修改 reduce()方法。当 Map 和 Reduce 功能都测试通过后,可以使用 MapReduceDriver 对 MapReduce 作业进行集成测试,测试代码如下。

```
public class WordCountTest {
   private Map map;
   private Reduce reduce;
   private MapReduceDriver driver;

   @Before
    public void init(){
      map=new WordCountMap();
     reduce=new WordCountReduce();
      driver=new MapReduceDriver(map,reduce);
    }

     @Test
    public void test()throws RuntimeException, IOException{
       String line="Hello Hadoop";
       driver .withInput("",new Text(line))
```

```
                    .withOutput(new Text("Hello"),new IntWritable(1))
                    .withOutput(new Text("Hadoop"),new IntWritable(1))
                    .runTest();
        }
}
```

通过 MapReduceDriver 的 withInput 构造 Map 的输入键值，通过 withOutput 构造 Reduce 的输出键值，来测试字数统计功能。单元测试和集成测试对程序开发是至关重要的，可以很快地发现并定位问题，从而保证了代码质量。

6. 打包发布

编写完 MapReduce 程序并进行测试通过后，下一步就要对 MapReduce 程序进行打包和集群运行。MapReduce 程序的打包过程如下。

在 Eclipse 的 IDE 中，选择 File→Export...命令，会出现导出资源选项卡，如图 5.18 所示。

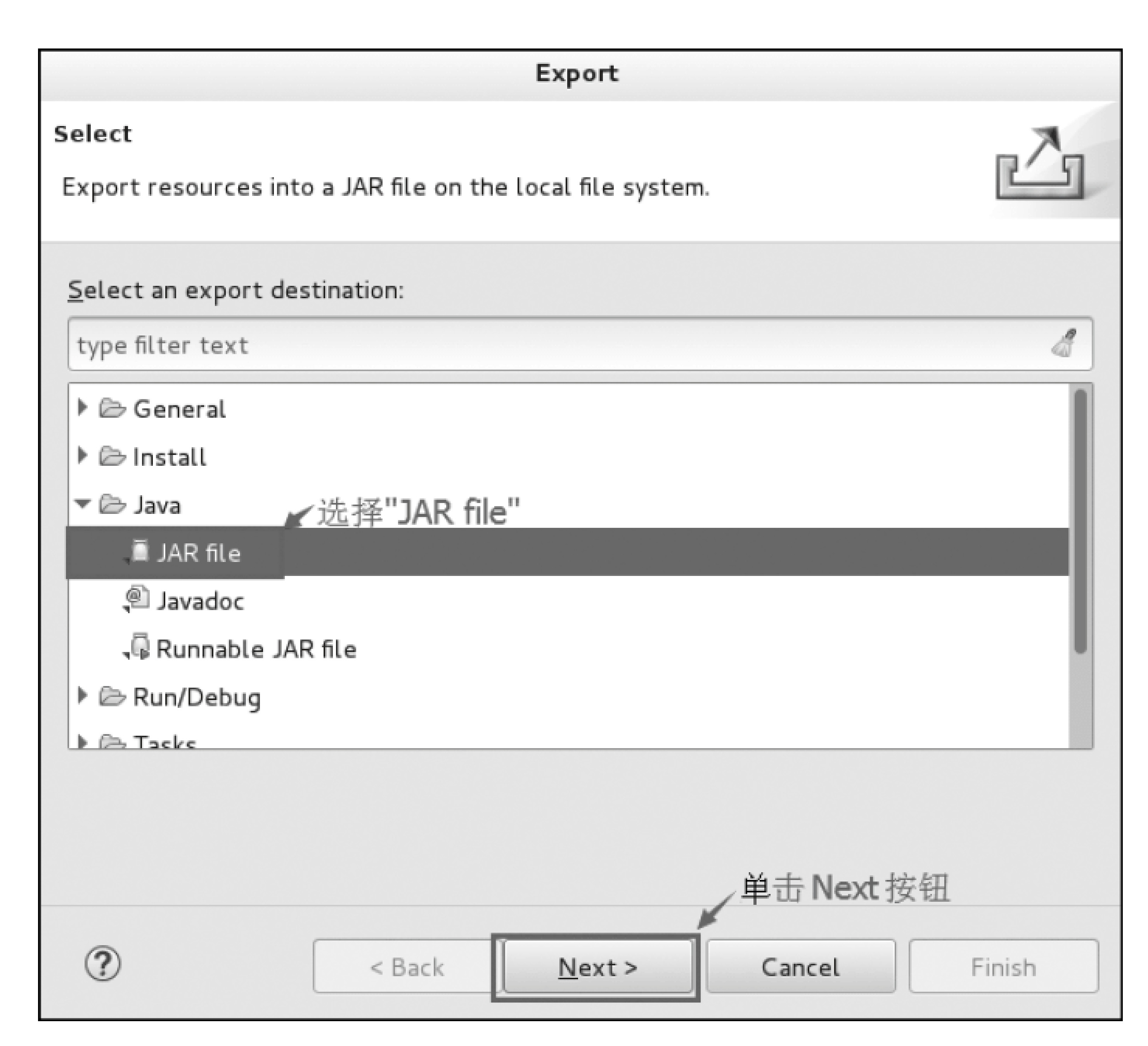


图 5.18　导出资源选项卡

单击 Next 按钮之后，出现如图 5.19 所示的选项卡。

在如图 5.19 所示的选项卡中，可以单击 Next 按钮或 Finish 按钮直接完成，生成名为 WordCount.jar 的文件，即 MapReduce 应用程序。最后，将 WordCount.jar 文件在 Hadoop 集群上运行即可，运行命令如下。



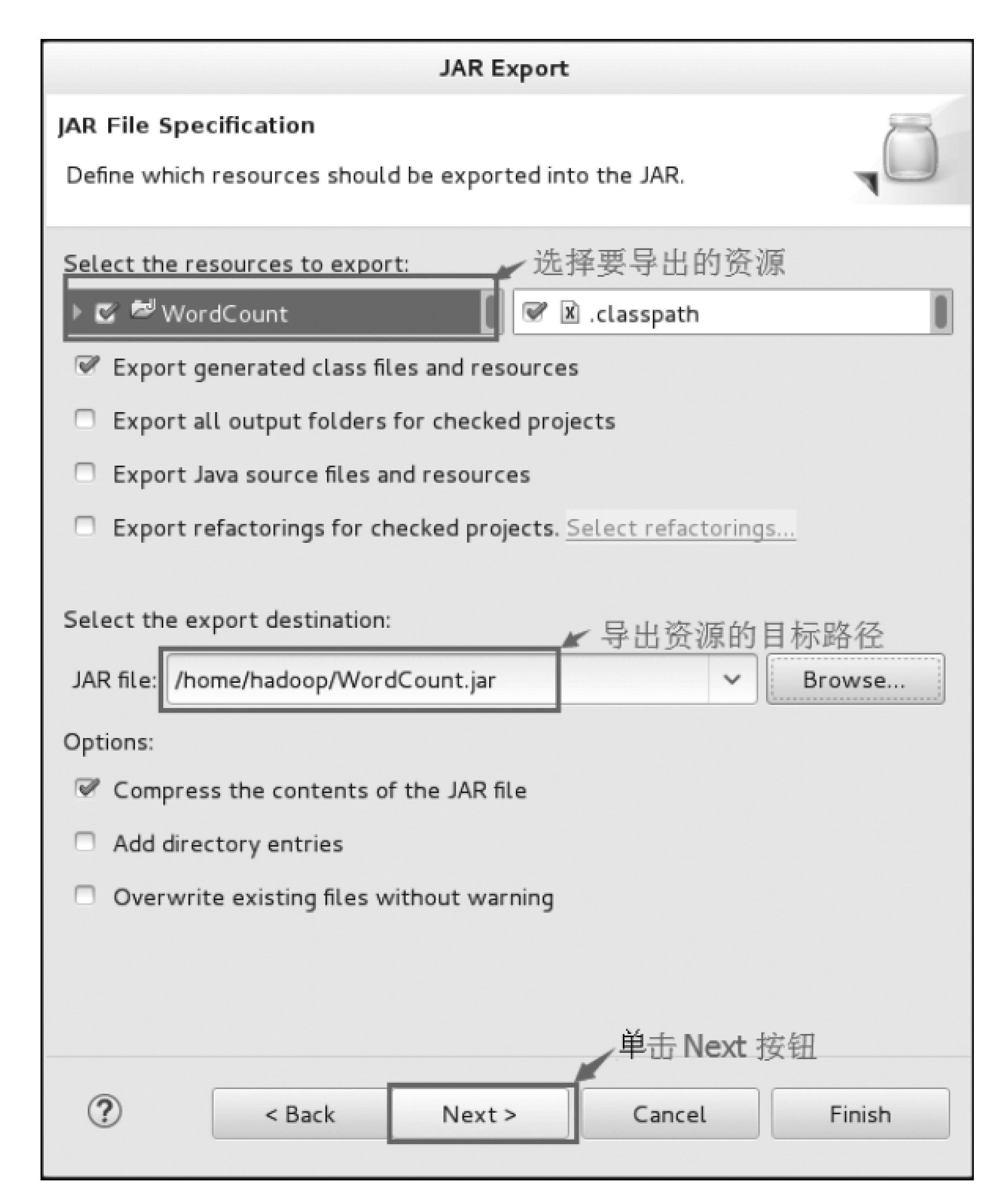


图 5.19　JAR 导出选项卡

```
//由于声明了 package ,所以在命令中要将 org.apache.hadoop.mapred 写完整
//input 为 in1 和 in2 文件所在的目录
//output 为 MapReduce 输出结果
[hadoop@ master1 dfs]$ hadoop jar /home/hadoop/WordCount.jar  org.apache.hadoop.mapred.WordCount /
input /output
```

正确输出的内容如下。

```
15/03/05 01:15:56 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
15/03/05 01:15:57 INFO client.RMProxy: Connecting to ResourceManager at /0.0.0.0:8032
15/03/05 01:15:59 INFO mapred.FileInputFormat: Total input paths to process : 2
15/03/05 01:15:59 INFO mapreduce.JobSubmitter: number of splits:2
15/03/05 01:16:00 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_1425536083584_0001
15/03/05 01:16:01 INFO impl.YarnClientImpl: Submitted application application_1425536083584_0001
15/03/05 01:16:01 INFO mapreduce.Job: The url to track the job: http://master1.hadoop:8088/proxy/
application_1425536083584_0001/
```

```
15/03/05 01:16:01 INFO mapreduce.Job: Running job: job_1425536083584_0001
15/03/05 01:16:16 INFO mapreduce.Job: Job job_1425536083584_0001 running in uber mode : false
15/03/05 01:16:16 INFO mapreduce.Job:  map 0% reduce 0%
15/03/05 01:16:44 INFO mapreduce.Job:  map 100% reduce 0%
15/03/05 01:16:58 INFO mapreduce.Job:  map 100% reduce 100%
15/03/05 01:16:59 INFO mapreduce.Job: Job job_1425536083584_0001 completed successfully
15/03/05 01:16:59 INFO mapreduce.Job: Counters: 49
        File System Counters
                FILE: Number of bytes read=78
                FILE: Number of bytes written=317135
                FILE: Number of read operations=0
                FILE: Number of large read operations=0
                FILE: Number of write operations=0
                HDFS: Number of bytes read=222
                HDFS: Number of bytes written=31
                HDFS: Number of read operations=9
                HDFS: Number of large read operations=0
                HDFS: Number of write operations=2
        Job Counters
                Launched map tasks=2
                Launched reduce tasks=1
                Rack-local map tasks=2
                Total time spent by all maps in occupied slots (ms)=48215
                Total time spent by all reduces in occupied slots (ms)=10630
                Total time spent by all map tasks (ms)=48215
                Total time spent by all reduce tasks (ms)=10630
                Total vcore-seconds taken by all map tasks=48215
                Total vcore-seconds taken by all reduce tasks=10630
                Total megabyte-seconds taken by all map tasks=49372160
                Total megabyte-seconds taken by all reduce tasks=10885120
        Map-Reduce Framework
                Map input records=4
                Map output records=8
                Map output bytes=78
                Map output materialized bytes=84
                Input split bytes=176
                Combine input records=8
                Combine output records=6
                Reduce input groups=4
                Reduce shuffle bytes=84
                Reduce input records=6
```

```
                Reduce output records=4
                Spilled Records=12
                Shuffled Maps=2
                Failed Shuffles=0
                Merged Map outputs=2
                GC time elapsed(ms)=475
                CPU time spent(ms)=3240
                Physical memory(bytes)snapshot=419872768
                Virtual memory(bytes)snapshot=2572558336
                Total committed heap usage(bytes)=256843776
        Shuffle Errors
                BAD_ID=0
                CONNECTION=0
                IO_ERROR=0
                WRONG_LENGTH=0
                WRONG_MAP=0
                WRONG_REDUCE=0
        File Input Format Counters
                Bytes Read=46
        File Output Format Counters
                Bytes Written=31
```

MapReduce 程序运行成功后，可以通过以下命令查看运行结果。

```
[hadoop@master1 dfs]$ hadoop fs -ls /output
Found 2 items
-rw-r--r-- 3 hadoop supergroup 0 2015-03-05 01:16 /output/_SUCCESS
-rw-r--r-- 3 hadoop supergroup 31 2015-03-05 01:16 /output/part-00000
[hadoop@master1 dfs]$ hadoop fs -cat /output/part-00000
Bye     2
Hadoop  2
Hello   2
World   2
```

也可以在 Web UI 界面中查看所提交的作业执行情况信息(http://master1.hadoop:8088)，展现了每个 Job 使用的 Map/Reduce 的数量、作业提交时间、作业启动时间、作业完成时间、Job ID、提交人 User、队列等信息，如图 5.20 所示。

5.6.2 实例解析

在编写 MapReduce 应用程序时，如果不理解 MapReduce 作业的运行机制，开发者将无法认识和开发 MapReduce 程序。下面通过上面的 WordCount 实例来介绍 MapReduce 作业的处理过程，如图 5.21 所示。

Logged in as: dr.who

hadoop

All Applications

- Cluster
  - About
  - Nodes
  - Applications
    - NEW
    - NEW_SAVING
    - SUBMITTED
    - ACCEPTED
    - RUNNING
    - FINISHED
    - FAILED
    - KILLED
  - Scheduler
- Tools

Cluster Metrics

提交的作业数量

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCores Used	VCores Total	VCores Reserved	Active Nodes	Decommissioned Nodes	Lost Nodes	Unhealthy Nodes	Rebooted Nodes
1	0	0	1	0	0 B	8 GB	0 B	0	8	0	1	0	0	0	0

Show 20 entries　　　作业起始时间　作业结束时间　当前作业状态　作业最终状态　Search:

ID	User	Name	Application Type	Queue	StartTime	FinishTime	State	FinalStatus	Progress	Tracking UI
application_1425536083584_0001	hadoop	wordcount	MAPREDUCE	default	Thu, 5 Mar 2015 06:16:01 UTC	Thu, 5 Mar 2015 06:16:57 UTC	FINISHED	SUCCEEDED		History

作业处理进程

Showing 1 to 1 of 1 entries　　First Previous 1 Next Last

(a)

Application Overview

User:	hadoop
Name:	wordcount
Application Type:	MAPREDUCE
Application Tags:	
State:	FINISHED
FinalStatus:	SUCCEEDED
Started:	5-Mar-2015 01:16:01
Elapsed:	55sec
Tracking URL:	History
Diagnostics:	

Application Metrics

Total Resource Preempted:	<memory:0, vCores:0>
Total Number of Non-AM Containers Preempted:	0
Total Number of AM Containers Preempted:	0
Resource Preempted from Current Attempt:	<memory:0, vCores:0>
Number of Non-AM Containers Preempted from Current Attempt:	0
Aggregate Resource Allocation:	191790 MB-seconds, 123 vcore-seconds

ApplicationMaster

Attempt Number	Start Time	Node	Logs
1	5-Mar-2015 01:16:01	Master1.Hadoop:8042	logs

(b)

图 5.20　MapReduce 作业运行情况

1. 输入数据

将 in1 和 in2 文件上传到 HDFS 中的 input 目录下，作为 MapReduce 应用程序的输入数据，如图 5.21(a)所示。其中，这两个文件的内容如下。

```
#in1
Hello World
Hello Hadoop

#in2
Bye World
Bye Hadoop
```

2. 文件分割

把输入的 in1 和 in2 文件通过 InputFormat 类切分成多个 splits。由于测试用的 in1 和 in2 文件较小，所以一个文件为一个 split，并通过 LineRecorderReader 将其中的每一行解析成＜key，value＞对作为 Map 的输入，如图 5.21(b)所示。其中，key 为该行在文本中的偏移量，value 值为这一行的内容。经过 InputFormat 类处理之后，in1 文件和 in2 文件分别形成了两个＜key，value＞对。如在 in1 文件中，第一对中 key 值为 0，是因为“Hello”单词位于文件头；第二对中 key 值为 12，是因为下一行的首单词“Hello”相对整个文本处于第 12 的位置。

3. Map 处理

将分割好的＜key，value＞对作为 map()方法的输入，然后由用户定义的 map()方法进行 Map 处理，生成新的＜key，value＞对，而且 Map 端会将这些结果按照 Key 值进行分组，如图 5.21(c)所示 map()方法的输出结果。

4. Combine 过程

得到 map()方法输出的＜key，value＞对之后，执行 Combine 过程，合并中间结果具有相同 key 值的键值，得到 Map 端的最终输出结果，如图 5.21(d)所示。

5. Reduce 处理

首先，Reduce 端接收到来自 Map 端的数据后，对数据进行排序，如图 5.21(e)所示。然后，再交由用户自定义的 reduce()方法进行处理，得到新的＜key，value＞对，作为 WordCount 的输出结果，如图 5.21(f)所示。

到此，MapReduce 的 WordCount 实例的基本过程已经解析完成，请读者通过该实例深入理解 MapReduce，达到触类旁通为止。

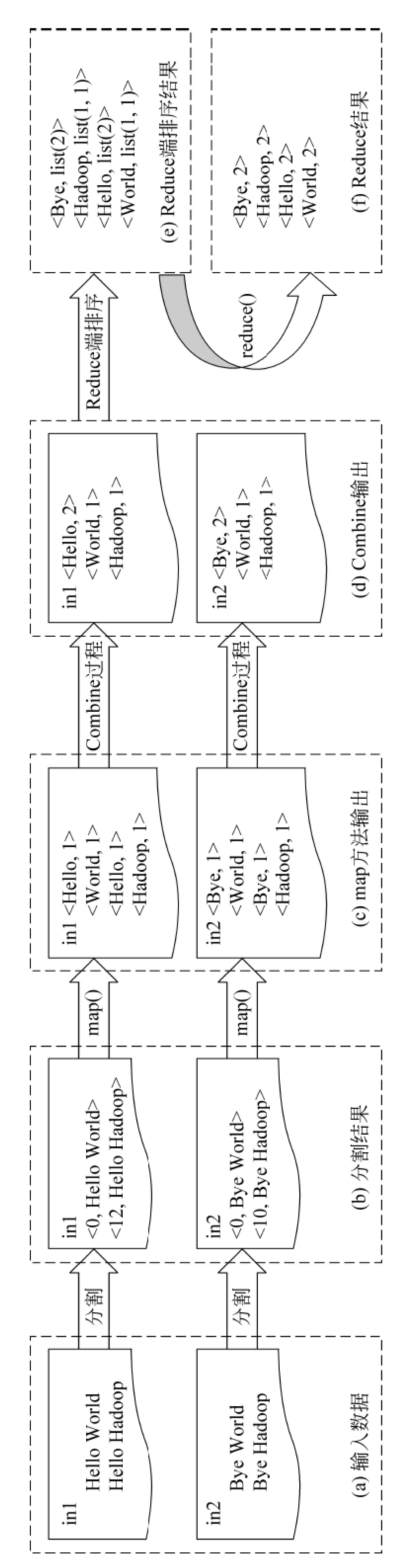


图 5.21 WordCount 处理过程

第6章 资源管理框架 YARN

资源管理框架 YARN(Yet Another Resource Negotiator)[67]是 Hadoop 2.0生态系统中新引入的Hadoop统一资源管理系统，是为了实现 Hadoop 集群共享、可伸缩性和可靠性而设计的。该框架采用了一种分层的集群框架思想，将集群资源管理与数据处理分离，并具有一定的通用性。本章将重点介绍 YARN 的产生背景、体系构架，以及运行在 YARN 上的计算框架等内容。

6.1 YARN 概述

最初的 Hadoop 1.0 生态系统的主要设计目的是处理和分析海量的日志数据和其他半结构化数据。因此，Hadoop 1.0 生态系统的应用场景主要是针对大规模的数据密集型计算，其应用场景比较单一，仅支持 MapReduce 的批处理计算方式，这种单一的 MapReduce 编程模型限制了开发者在多种场景下的海量数据处理和应用。而且，Hadoop 1.0 生态系统采用了一种紧耦合的资源管理方式，即 MapReduce 编程模型和资源管理的紧耦合，迫使开发者滥用了 MapReduce 编程模型，并且中心化的作业控制流也影响了系统的可拓展性。随着 Hadoop 集群的规模越来越大，数据量逐日剧增，作业的数量也显著增加，对 Hadoop 的可拓展性提出了更高的要求。

Hortonworks 的创始人兼架构师 Arun Murthy 基于上述的 Hadoop 设计缺陷和日益增长的业务需求开始重新设计了 Hadoop 架构，即 Hadoop 2.0 架构。Hadoop 2.0 生态系统中的 YARN 架构是在 MapReduce(MRv1)基础上演化而来的，YARN 的最初设计是为了修复 MapReduce(MRv1)中的各种局限性，并对可伸缩性、可靠性和集群利用率进行了提升。YARN 修复了 MapReduce(MRv1)中的局限性，主要体现在以下几点。

(1) 可扩展性。YARN 提高了 MRv1 的可扩展性。Hadoop 1.0 架构受到了 JobTracker 的高度约束，JobTracker 同时负责资源管理和作业控制两个功能；新的 YARN 架构打破了这种约束，把 JobTracker 的两个功能分成了两个独立的服务程序，即全局的资源管理(ResourceManager)和针对每个应用的应用 Master(ApplicationMaster)。其中，ResourceManager 承担了 MRv1 中 TaskTracker 的一些角色；ResourceManager 承担了 MRv1 中 JobTracker 的角色。同时，YARN 框架支持不同的编程模型，如 Spark、Storm、Tez 等。

(2) 可靠性。Hadoop 1.0 生态系统主要由 MapReduce(MRv1)和 HDFS 组成，这两个组件的设计缺陷是单点故障，即 MRv1 的 JobTracker 和 HDFS 的 NameNode 两个核

心服务均存在单点故障。Hadoop 2.0 生态系统中引入了 YARN 作为资源管理系统，使得 Hadoop 不再局限于 MapReduce 计算，而且支持多样化的计算框架，提高了计算能力的可靠性。虽然 YARN 也存在单点故障[68]（主要是 ResourceManager 单点故障），但是由于每个作业独立使用一个作业跟踪器（ApplicationMaster），彼此互不影响，当备用 ResourceManager 启动后，会从共享存储系统中重新读取这些 Application 的元数据信息，并重新提交这些 Application。因此，解决 YARN 的单点故障要比 Hadoop 1.0 生态系统中 NameNode 和 HDFS 的单点故障容易得多，并且可靠性更高。

（3）资源利用率。Hadoop 1.0 生态系统中的资源管理由资源表示模型和资源分配模型组成，并采用了基于槽位（Slot）的粗粒度资源划分单位组织各节点上的资源（CPU、内存、磁盘空间等资源）。这种基于槽位的资源划分粒度过于粗糙，往往会造成节点资源利用率过高或过低，而且资源之间是采用 JVM 的隔离机制，这种资源隔离机制无法实现真正的资源隔离，会造成同一个节点上任务之间干扰严重，影响资源利用率。而 YARN 中的资源管理是由 ResourceManager 和 NodeManager 共同完成的。其中，ResourceManager 中的调度器负责资源的分配；NodeManager 则负责资源的供给和隔离。ResourceManager 的资源分配不再基于槽位（Slot）的粗粒度划分，而是让任务直接向调度器申请自己所需要的资源，提高了资源利用率；NodeManager 按照需求为任务提供相应的资源，并保证这些资源的独占性，为任务运行提供基本保证。

（4）支持多种计算框架。在 Hadoop 1.0 生态系统中，只有一种计算框架，即 MapReduce（MRv1），无法满足多场景应用的需求。在 Hadoop 2.0 生态系统中的 YARN 支持多种计算框架，除支持默认的 MapReduce（MRv2）计算框架外，还支持内存计算框架 Spark、流式计算框架 Storm、DAG 计算框架 Tez 等。

总之，YARN 是一种通用的 Hadoop 资源管理框架，犹如 Hadoop 的“操作系统”。该框架不仅支持离线的数据处理（MapReduce），还支持如实时数据处理（Spark）、流式数据处理（Storm）、图数据处理（Tez）等计算框架。随着 YARN 的不断成熟和稳定，各类应用程序将运行在一个以 YARN 为核心的集群中进行统一资源管理和调度，从而使得应用程序和服务部署将更加简单，更有效地进行集群资源管理和隔离等。

6.2 YARN 体系架构

YARN 是 Hadoop 2.0 生态系统中新引入的资源管理框架，总体上仍是 Master/Slave 结构，该框架主要由 ResourceManager、NodeManager、ApplicationMaster 和 Container 组件构成，如图 6.1 所示。

其中，ResourceManager 相当于 Master，主要负责对各个 NodeManager 上的资源进行统一管理、调度和状态监控；NodeManager 相当于 Slave，主要负责每个节点上的资源管理、任务执行、心跳上报等；每个应用有一个 ApplicationMaster，主要负责应用程序的申请资源、监控任务运行和容错等；Container 是对任务运行环境的抽象。

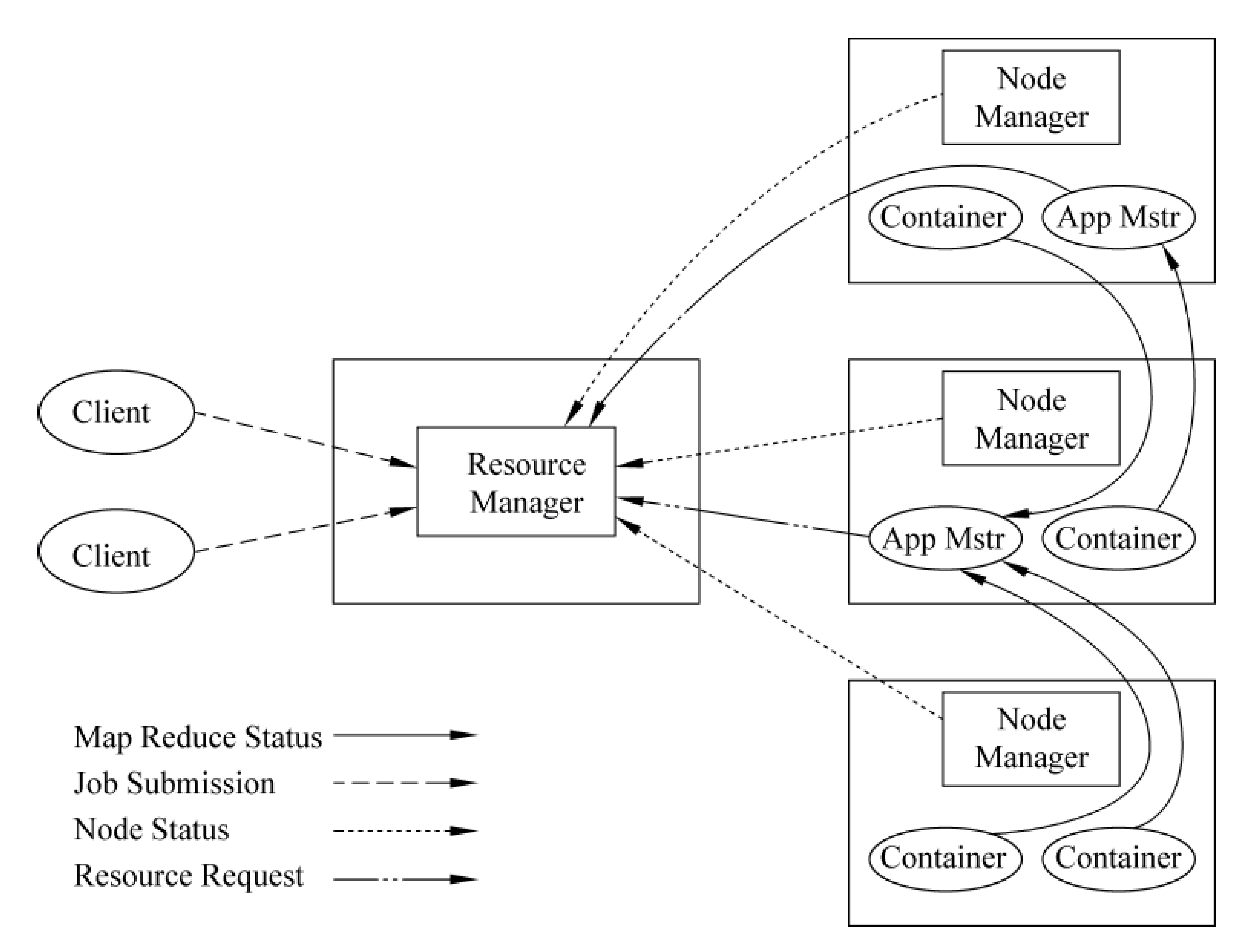


图 6.1　YARN 基本架构

6.2.1　ResourceManager

ResourceManager(简称 RM)是 YARN 的 Master,具有中心化的全局资源视图,主要负责集群中所有资源的统一管理和分配,它接收来自各个 NodeManger 的资源汇报信息,并进行相应的任务调度、资源管理和状态监控。ResourceManager 根据不同的应用需求、调度优先级和资源情况等,动态调整资源。ResourceManger[69] 的主要组成如图 6.2 所示。

从图 6.2 中可以看出,ResourceManager 主要提供了用户交互管理、ApplicationMaster 管理、NodeManager 管理、Application 管理、资源分配管理、安全管理等功能。

1. 用户交互管理

ResourceManager 分别针对不同的用户,实现了对外提供不同的服务管理,如对于普通用户和管理员用户分别对应 ClientRMService 和 AdminService。

1) ClientRMService

ClientRMService 是为普通用户提供的服务,它会处理来自客户端的各种 RPC 请求,如提交应用程序、终止应用程序、获取应用程序运行状态等。如果需要了解该类的具体实现功能,可以查看 Hadoop 源码(以 Hadoop 2.6.0 为例)目录下的 hadoop-yarn-server-resourcemanager 工程中的 org.apache.hadoop.yarn.server.resourcemanager.ClientRMService 类。

2) AdminService

AdminService 是为防止大量普通用户请求使用管理员权限而专门为管理员提供的

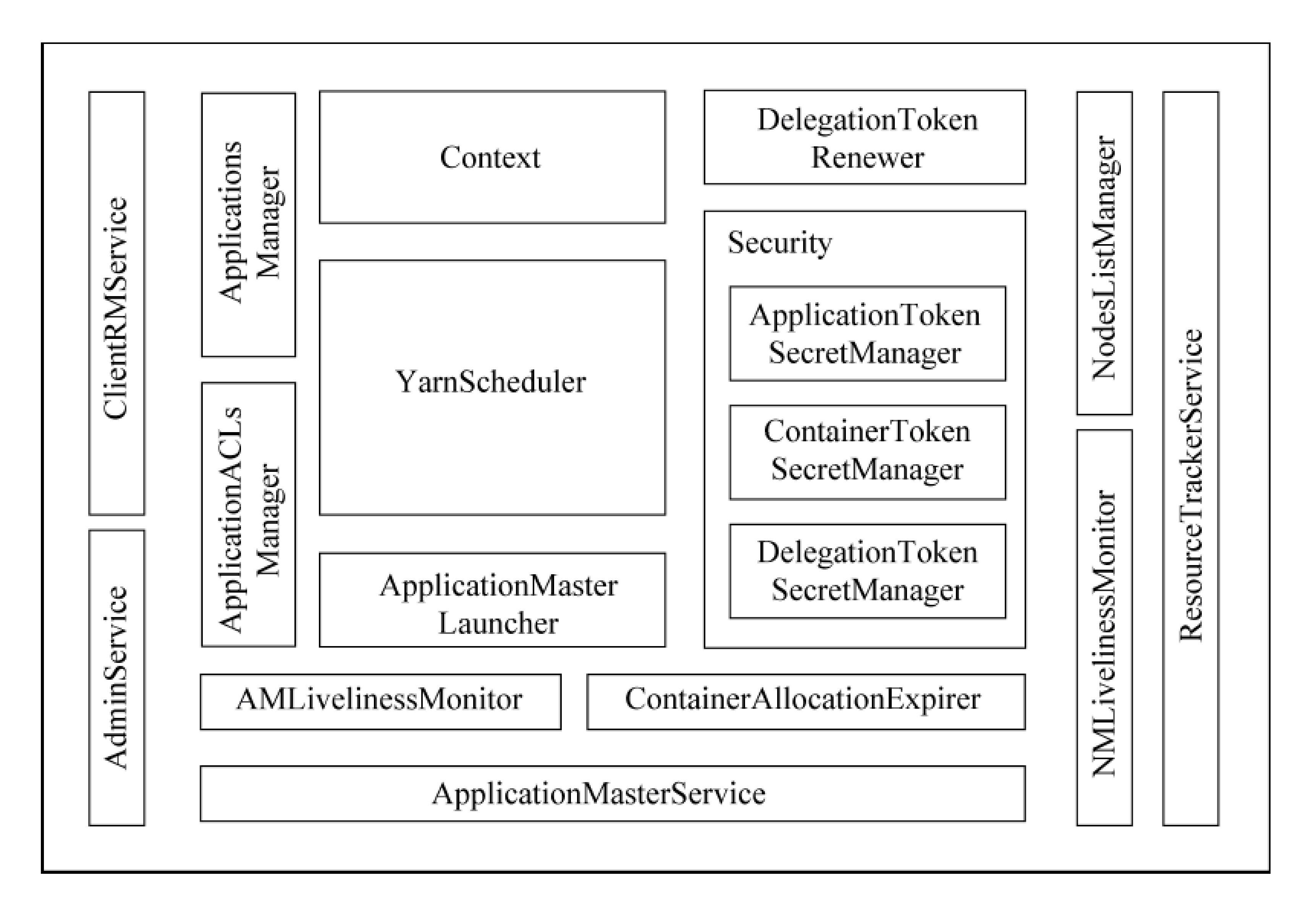


图 6.2　ResourceManager 基本组成

(图片来源：http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/)

一套独立的服务接口。管理员可通过该接口实现 Hadoop 集群的管理，如动态更新节点列表、更新 ACL 列表、更新队列信息等。如果需要了解该类的具体实现功能，可以查看 Hadoop 源码(以 Hadoop 2.6.0 为例)目录下的 hadoop-yarn-server-resourcemanager 工程中的 org.apache.hadoop.yarn.server.resourcemanager.AdminService 类。

2. ApplicationMaster 管理

ApplicationMaster 的管理主要是通过 ResourceManager 内部的三个组件来完成的，即 AMLivelinessMonitor、ApplicationMasterLauncher 和 ApplicationMasterService 组件。

1) AMLivelinessMonitor

AMLivelinessMonitor 主要用来监控 ApplicationMaster 是否存活，周期性地遍历集群中的所有 ApplicationMaster，如果一个 ApplicationMaster 在一定时间内(默认为 10min，可以通过参数 yarn.am.liveness-monitor.expiry-interval-ms 配置)未汇报心跳信息，则认为该 ApplicationMaster 已无心跳，并把该 ApplicationMaster 上所有正在运行的 Container 置为失败并释放资源，ApplicationMaster 会重新为它申请资源，并且会重新分配到另外节点上启动它(ApplicationMaster 启动尝试次数由参数 yarn.resourcemanager.am.max-attempts 控制，默认为 2)。

2) ApplicationMasterLauncher

ApplicationMasterLauncher 是以线程池方式实现的一个事件处理器，主要处理 AMLauncherEvent 类型的事件，包括启动(LAUNCH)和清除(CLEANUP)一个 ApplicationMaster 的事件。当接收到 LAUNCH 类型的事件时，ApplicationMasterLauncher

会与对应的 NodeManager 进行通信，并启动该 ApplicationMaster 所需要的如启动命令、JAR 包、环境变量等信息，并启动 ApplicationMaster；当接收到 CLEANUP 类型事件时，ApplicationMasterLauncher 会立刻与对应的 NodeManager 进行通信，从而要求 NodeManager 杀死该 ApplicationMaster，并释放资源。

3）ApplicationMasterService

ApplicationMasterService 主要负责处理来自 ApplicationMaster 的心跳请求和 Application 的注册与清理请求。ApplicationMasterService 处理 ApplicationMaster 的心跳请求是周期性行为，ApplicationMaster 向 ApplicationMasterService 发送心跳请求包，包括请求资源类型的描述、待释放的 Container 列表等，ApplicationMasterService 处理心跳后的应答信息包括新分配的 Container、失败的 Container 等信息。ApplicationMasterService 处理注册请求是在 Application 启动完成后发生的，其中注册请求信息包括 ApplicationMaster 所在的节点、RPC 端口、Tracking URL 等信息；而清理请求是在 ApplicationMaster 运行结束后发生的，主要是回收释放资源。

3. NodeManager 管理

NodeManager 的管理主要是通过 ResourceManager 内部的三个组件来完成的，即 NMLivelinessMonitor、ResourceTrackerService 和 NodeListManager 组件。

1）NMLivelinessMonitor

NMLivelinessMonitor 功能同 AMLivelinessMonitor，只不过 AMLivelinessMonitor 用来监控 ApplicationMaster，而 NMLivelinessMonitor 是用来监控 NodeManager。另外，该服务会周期性地遍历集群中 NodeManager 的时间默认为 10，可以通过参数 yarn. nm. liveness-monitor. expiry-interval-ms 配置。

2）ResourceTrackerService

ResourceTrackerService 是 RPC 协议 ResourceTracker 的一个实现，它作为一个 RPC Server 端接收 NodeManager 的 RPC 请求，主要功能是注册 NodeManager 和处理心跳信息。

3）NodeListManager

NodeListManager 主要负责白名单（exclude 列表）和黑名单（include 列表）的管理功能。其中，白名单和黑名单可以通过 yarnresourcemanager. nodes. include-path 和 yarnresourcemanagernodes. exclude-path 来指定。黑名单列表中的 NodeManager 不能和 ResourceManager 直接通信，而白名单列表中的 NodeManager 可以和 ResourceManager 直接通信。

4. Application 管理

Application 的管理主要是通过 ResourceManager 内部的三个组件来完成的，即 ApplicationACLsManager、RMAppManager 和 ContainerAllocationExpirer 组件。

1）ApplicationACLsManager

ApplicationACLsManager 主要用于维护每个 Application 的 ACLs 列表，即管理

Application 的查看和修改权限。其中,查看主要是指查看 Application 基本信息,而修改主要是修改 Application 的优先级、杀死 Application 等。

2) RMAppManager

RMAppManager 在 ResourceManager 中的作用是管理所有的 Application,如提交、完成 Application,恢复一组 Applications 等。

3) ContainerAllocationExpirer

ContainerAllocationExpirer 主要负责保证所有分配给 ApplicationMaster 的 Container 都会被相应的 NodeManager 启动起来,从而提高整个集群的利用率。

5. 资源分配管理

ResourceManager 的资源分配管理主要由 YarnScheduler 来完成,YarnScheduler 是基于 Application 资源需求的分配策略,将资源分配给正在运行的 Application,如 CPU、内存、网络、磁盘等。在 Hadoop 1.0 生态系统中,YarnScheduler 默认的资源分配策略是 FIFO 实现的(FIFO Scheduler),但还可以使用其他的资源分配策略,如 Fair Scheduler 和 Capacity Scheduler(Hadoop 2.0 生态系统中 YarnScheduler 的默认实现方式)。这三种资源分配策略的比较如表 6.1 所示。

表 6.1 三种资源分配策略比较

	FIFO Scheduler	Capacity Scheduler	Fair Scheduler
特点	最简单的 FIFO 策略	多用户情况下,最大化集群的吞吐率和利用率	多用户情况下,强调用户公平地贡献资源
队列组织方式	单队列	树状组织队列(子队列和父队列存在继承关系)	树状组织队列(子队列和父队列不存在继承关系)
资源限制	无	父子队列之间有容量关系	每个子队列有最小共享量、最大资源量和最大活跃应用数量
队列排序算法	无	按照队列的资源使用量最小的优先	根据公平排序算法排序
本地优先分配	支持	支持	支持
延迟调度	不支持	不支持	支持
资源抢占	不支持	不支持	支持

6. 安全管理

ResourceManager 的安全管理机制主要由 DelegationTokenSecretManager、ContainerTokenSecretManager、DelegationTokenRenewer 和 ApplicationTokenSecretManager 等组件完成。DelegationTokenRenewer 主要负责更新已提交的 Applications 的 tokens,直到这个 tokens 不再被更新为止;DelegationTokenSecretManager、ContainerTokenSecretManager 和 ApplicationTokenSecretManager 默认是采用 Kerberos 认证的,用于认证或授权各种 RPC

接口的请求。

6.2.2　NodeManager

NodeManager 主要负责每个节点上的资源管理、任务执行、心跳上报等。一方面，它会定时通过心跳信息向 ResourceManager 汇报本节点上的资源使用情况和各个 Container 的运行情况；另一方面，它会接收并处理来自 ApplicationMaster 的 Container 启动和停止的各种请求。NodeManger[70] 的主要组成如图 6.3 所示。

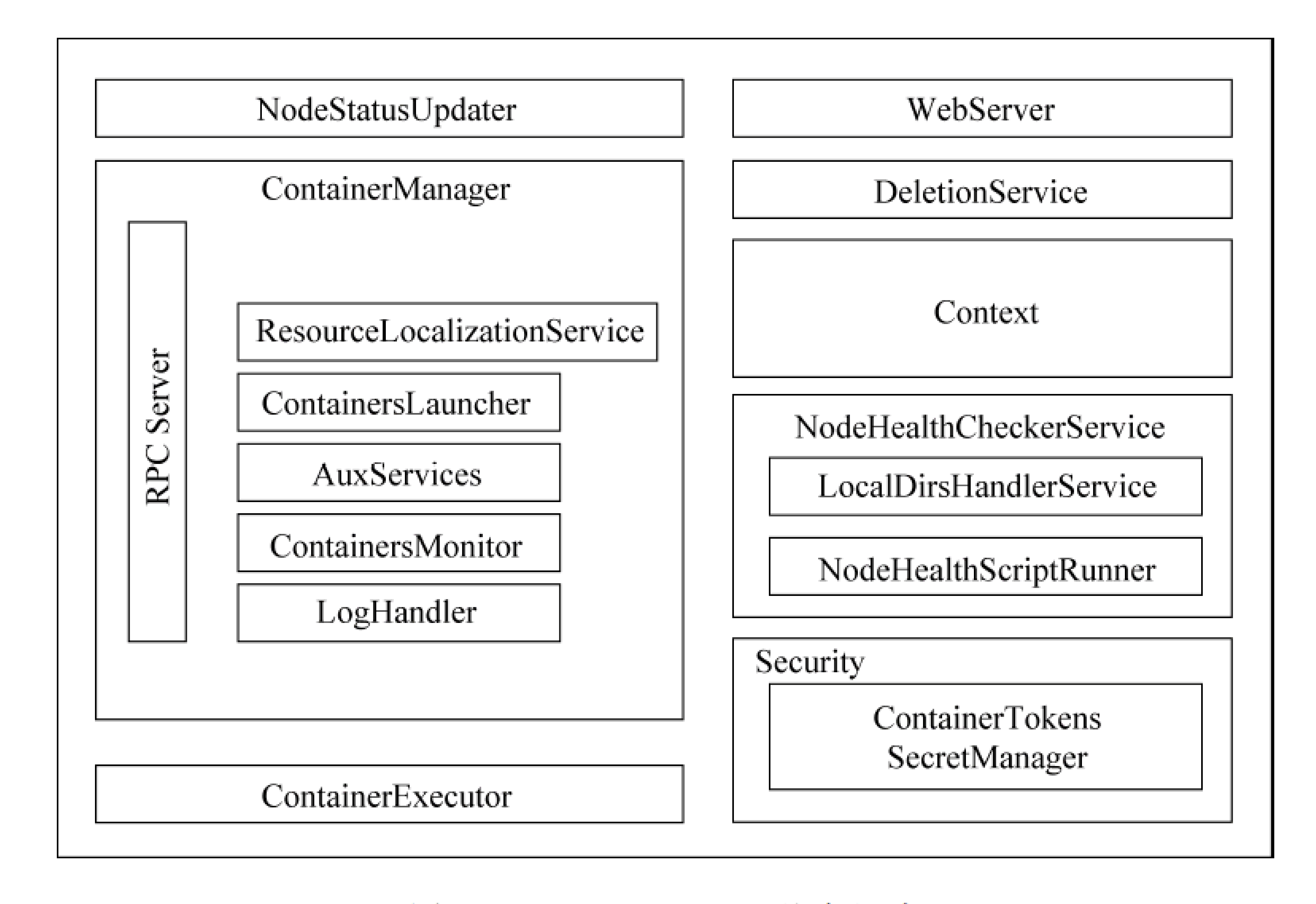


图 6.3　NodeManager 基本组成

（图片来源：http://hortonworks.com/blog/apache-hadoop-yarn-nodemanager/）

其中，NodeStatusUpdater 是 NodeManager 与 ResourceManager 之间的唯一通道，周期性地调用 RPC. nodeHeartbeat() 向 ResourceManager 汇报本节点上的资源使用情况、各个 Container 运行情况等信息；ContainerManager 是 NodeManager 的核心组件，由 RPC Server、ResourceLocalizationService、ContainersLauncher、AuxServices、ContainersMonitor 和 LogHandler 组件构成，每个组件负责一部分功能，以管理运行在该节点上的所有 Container；ContainerExecutor 用于与底层的操作系统进行交互，YARN 提供了两种 ContainerExecutor 的实现，即 DefaultContainerExecutor（默认实现方式）和 LinuxContainerExecutor（新引入的）；NodeHealthCheckerService 主要由 LocalDirsHandlerService 和 NodeHealthScriptRunner 组成，主要用于周期性地检查节点的健康情况，并提供可访问的 API；WebServer 主要用于展示该节点上所有应用程序和 Container 列表、节点健康相关的信息和 Container 产生的日志。

6.2.3　ApplicationMaster

ApplicationMaster 主要负责应用程序的申请资源、监控任务运行和容错等。每个 Application 都会有一个与之对应的 ApplicationMaster，ApplicationMaster 会为与之对应

的 Application 向 ResourceManager 申请资源、与 NodeManager 通信以启动或者停止任务、监控所有任务的运行情况,并且在任务失败的情下,重新为任务申请资源并且重启任务、负责推测任务的执行、当 ApplicationMaster 向 ResourceManager 注册后,ApplicationMaster 可以提供客户端查询作业进度信息等。

当前 YARN 自带了两个 ApplicationMaster 的实现,一个是用于演示 ApplicationMaster 编写方法的实例程序 distributedshell,另一个是运行 MapReduce 应用程序的 ApplicationMaster,即 MRAppMaster。其中,distributedshell 所实现的 ApplicationMaster 可以申请一定数目的 Container 以并行运行一个 Shell 命令或 Shell 脚本;MapReduce 所实现的 ApplicationMaster(MRAppMaster)主要由 ContainerAllocator、ContainerLauncher、RecoveryService、MRClientService、JobHistoryEventHandler 和 Speculator 等组件组成。其中,ContainerAllocator 用于向 ResourceManager 申请资源;ContainerLauncher 用于请求 NodeManager 启动 MRYarnChild;RecoveryService 用于重新恢复 MRAppMaster 的运行环境;JobHistoryEventHandler 用于处理 jobhistory 信息;Speculator 用于推测执行等。MRAppMaster 主要负责 MapReduce 作业的生命周期的管理,包括创建 MapReduce 作业、向 ResourceManager 申请资源、与 NodeManager 通信启动 Container、监控作业的运行状态、当任务失败时重启任务等,使得 MapReduce 计算框架可以运行于 YARN 之上。目前,运行在 YARN 上的计算框架除了默认实现的 MapReduce 之外,还有 Spark on YARN(内存计算框架)、Storm on YARN(实时/流式计算框架)、Tez on YARN(DAG 计算框架)等。用户也可以根据自己的实际需求出发,针对不同的编程模型实现自己的 ApplicationMaster,使之可以成为运行在 YARN 上的框架。

6.2.4 Container

Container 是系统资源动态分配的基本单位,是对系统资源的抽象。YARN 会为每个任务分配一个 Container,并且该任务只能够使用该 Container 中所描述的资源,如 CPU、内存、磁盘、网络等。Container 中所描述的资源是根据实际的 Application 需求而动态变化的,其生命周期主要包括启动、运行和资源回收的过程。

1. 启动 Container

在启动 Container 之前,需要初始化 Container 运行所需的环境,如创建对应 Container 目录、从 HDFS 下载 Container 运行所需的 Jar 包和可执行文件等。准备好 Container 运行环境之后,Container 的启动命令是由各个 ApplicationMaster 通过 RPC 通信向 NodeManager 发送请求指令,再由 ContainersLauncher 来完成 Container 的启动。

2. 运行 Container

Container 的运行是由 ApplicationMaster 向资源所在的 NodeManager 发送运行请求指令,再由 NodeManager 中的 ContainerExecutor 来完成运行的。

3. 资源回收

当任务执行完毕后，Container 需要对所占用的资源进行释放回收，这些是由 NodeManager 中的 ResourceLocalizationService 来完成的。

6.3　YARN 工作流程

在 YARN 上的应用程序按其生命周期的长短可分为长应用程序和短应用程序。其中，长应用程序是指在程序启动后将一直运行下去；短应用程序是指在程序启动后在有限时间内运行完成后，释放该应用程序所占的资源，从而方便 YARN 进行再分配。不论长应用程序还是短应用程序，YARN 的工作流程都一样，具体的工作流程如图 6.4 所示。

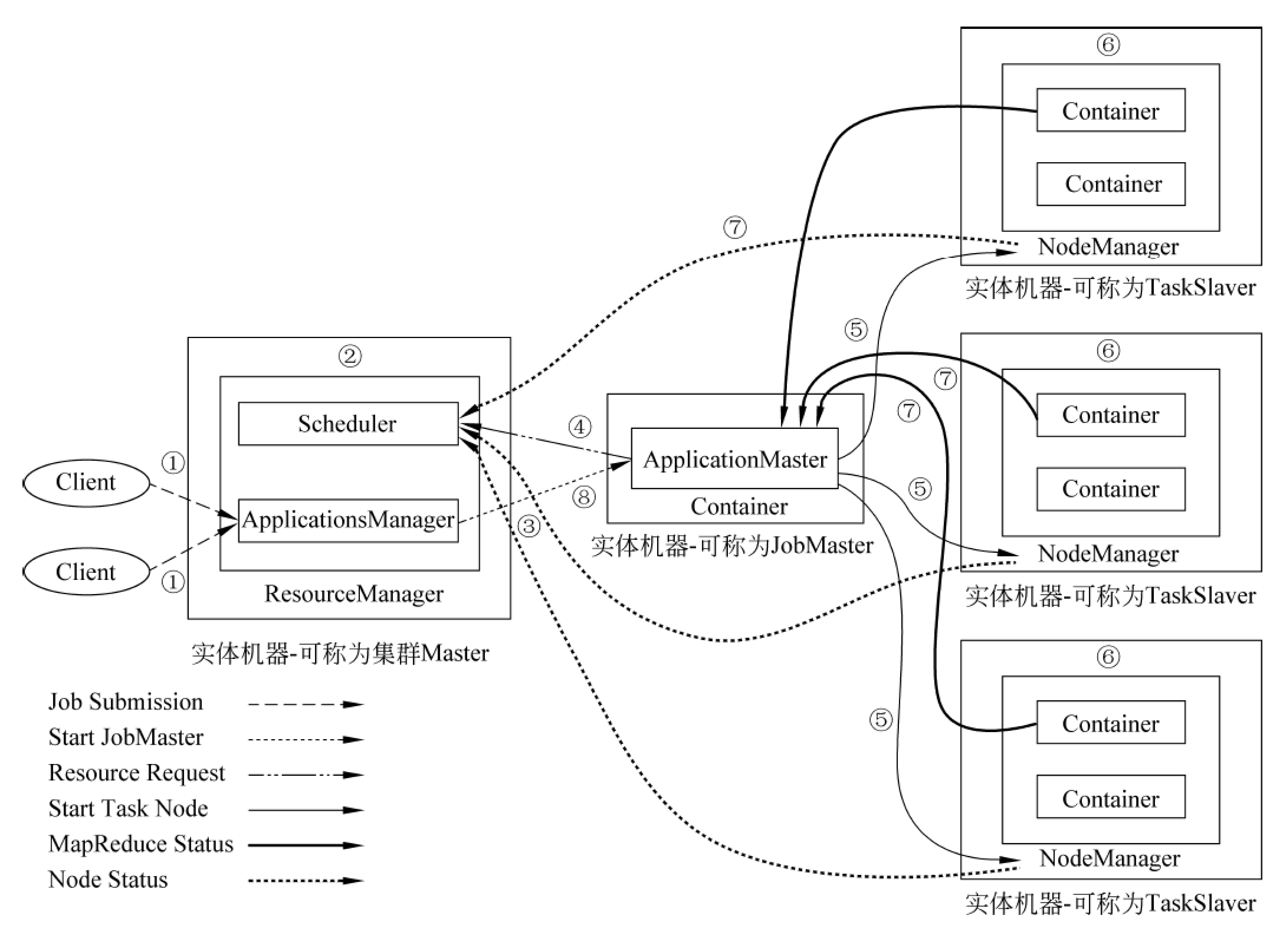


图 6.4　YARN 工作流程

(1) 由 Client 端向 YARN 提交应用程序到 ResourceManager 上，提交的内容包括 ApplicationMaster 程序、启动 ApplicationMaster 的命令、本身用户程序的内容等。

(2) ResourceManager 为应用程序 ApplicationMaster 申请资源，并分配第一个 Container。ApplicationMaster 获取到资源后，与对应的 NodeManager 通信，并要求该 NodeManager 在对应的 Container 中启动应用程序对应的 ApplicationMaster。

(3) ApplicationMaster 首先会向 ResourceManager 进行注册，来表明自己启动成

功。注册成功之后，就可以通过 ResourceManager 来管理 ApplicationMaster，而且用户也可以直接通过 ResourceManager 的 Web 客户端来查看应用程序的运行状态。然后 ResourceManager 将为各个任务申请资源，并监控它的运行状态，不断重复(4)～(8)，直到运行结束。

(4) ApplicationMaster 周期性地向 ResourceManager 中的 ResourceScheduler 申请资源，资源申请成功后再领取相应资源。

(5) 一旦 ApplicationMaster 申请到资源后，ApplicationMaster 会与申请到的 Container 所对应的 NodeManager 进行通信，并将启动命令交给 NodeManager，并要求 ApplicationMaster 在该 Container 中启动任务。

(6) NodeManager 为要启动的任务配置好运行环境，如环境变量、JAR 包、二进制程序等，并且将启动命令写在一个脚本中，通过该脚本在 Container 内执行用户提交的代码。

(7) 各个 Container 通过某个 RPC 协议向 ApplicationMaster 汇报自己的状态和进度，以便让 ApplicationMaster 随时掌握各个任务的运行状态，从而保证在任务失败时重新启动该任务。在应用程序运行过程中，用户也可以随时通过 RPC 向 ApplicationMaster 查询应用程序的当前运行状态。

(8) 当应用程序运行结束后，ApplicationMaster 向 ResourceManager 注销并关闭自己。

在 YARN 的整个工作流程中，ResourceManager 与 NodeManager 之间都是通过心跳保持联系的，NodeManager 会通过心跳信息向 ResourceManager 汇报自己所在节点的资源使用情况。

6.4 YARN 通信机制

YARN 中各组件之间的通信都需要 RPC 协议的支持，对于不同组件之间的通信，所需的 RPC 协议类型不同。YARN 所采用的是一种拉式(Pull)的通信模型，即任何两个需相互通信的组件之间仅有一个 RPC 协议；对于任何一个 RPC 协议，都是基于 Client/Server 模式，其中请求服务的一端为 Client，提供服务的一端为 Server，而且 Client 总是主动连接 Server。YARN 的通信形式有 Client(作业提交客户端)和 ResourceManager、Admin(管理员)和 ResourceManager、ApplicationMaster 和 ResourceManager、ApplicationMaster 和 NodeManager、NodeManager 和 ResourceManager 之间的通信，如图 6.5 所示(箭头指向的组件为 RPC Server，箭头另一端的组件为 RPC Client)。

在整个 YARN 框架中主要涉及 7 个 RPC 协议，分别是 ApplicationClientProtocol、MRClientProtocol、ContainerManagementProtocol、ApplicationMasterProtocol、ResourceTracker、ResourceLocalizerProtocol、TaskUmbilicalProtocol 协议。而在 YARN 中各组件之间进行通信用到的主要协议如下。

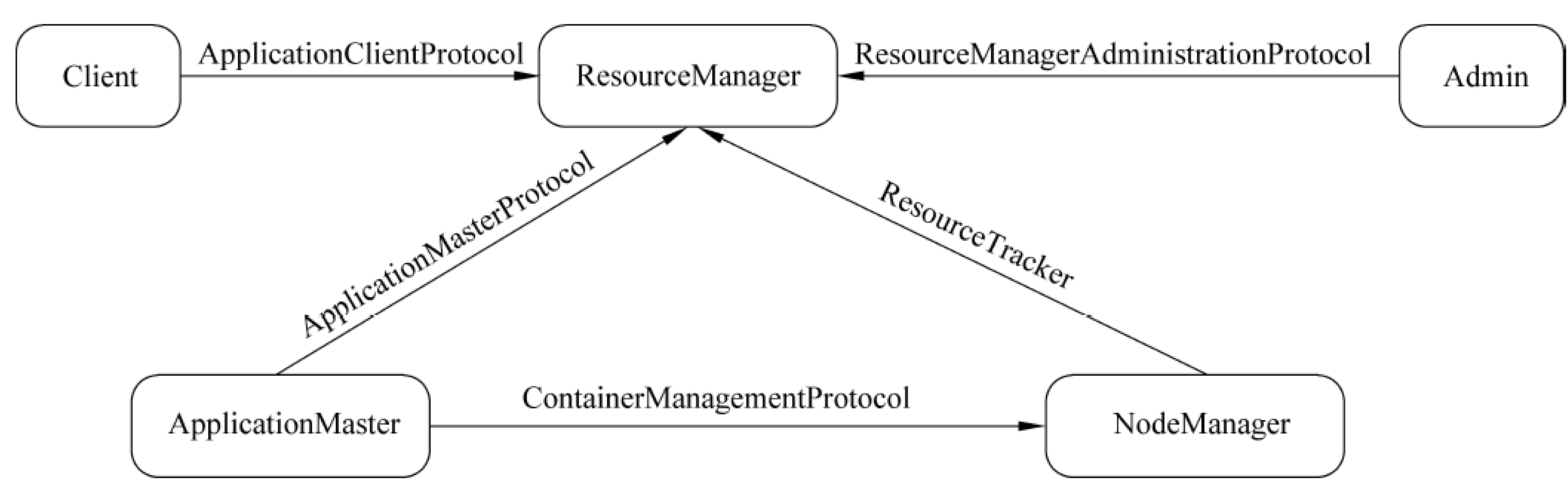


图 6.5 YARN 各组件之间的通信

1. ApplicationClientProtocol

ApplicationClientProtocol 通信协议主要用于作业提交客户端 Client 与 ResourceManager 之间的通信。其中，作业提交客户端 Client 为 RPC Client，ResourceManager 为 RPC Server。作业提交客户端 Client 通过该 RPC 协议提交应用程序，查询应用程序状态等。

2. ResourceManagerAdministrationProtocol

ResourceManagerAdministrationProtocol 通信协议主要用于管理员端 Admin 与 ResourceManager 之间的通信。其中，管理员端为 RPC Client，ResourceManager 为 RPC Server。管理员端 Admin 通过该协议更新系统配置文件，比如节点黑名单、用户队列权限等。

3. ApplicationMasterProtocol

ApplicationMasterProtocol 通信协议主要用于 ApplicationMaster 与 ResourceManager 之间的通信。其中，ApplicationMaster 为 RPC Client，ResourceManager 为 RPC Server。ApplicationMaster 通过该 RPC 协议向 ResourceManager 注册和撤销自己，并为各个任务申请资源。

4. ContainerManagementProtocol

ContainerManagementProtocol 通信协议主要用于 ApplicationMaster 与 NodeManager 之间的通信。其中，ApplicationMaster 为 RPC Client，NodeManager 为 RPC Server。ApplicationMaster 通过要求 NodeManager 启动或者停止 Container，获取各个 Container 的使用状态等信息。

5. ResourceTracker

ResourceTracker 通信协议主要用于 NodeManager 与 ResourceManager 之间的通信。其中，NodeManager 为 RPC Client，ResourceManager 为 RPC Server。NodeManager 通过该

RPC 协议向 ResourceManager 注册，并定时发送心跳信息汇报当前节点的资源使用情况和 Container 运行状况。

另外，为了提高 Hadoop 不同版本之间的兼容性和性能，YARN 的序列化框架采用了 Google 开源的 Protocol Buffers。YARN 中的所有 RPC 通信协议都是以 proto 文件的形式进行定义的。例如，applicationmaster_protocol. proto 文件定义了 ApplicationMaster 与 ResourceManager 之间的通信协议 ApplicationMasterProtocol，其文件内容如下。

```
option java_package="org.apache.hadoop.yarn.proto";
option java_outer_classname="ApplicationMasterProtocol";
option java_generic_services=true;
option java_generate_equals_and_hash=true;
package hadoop.yarn;

import "yarn_service_protos.proto";

service ApplicationMasterProtocolService {
  rpc registerApplicationMaster (RegisterApplicationMasterRequestProto) returns
  (RegisterApplicationMasterResponseProto);
  rpc finishApplicationMaster (FinishApplicationMasterRequestProto) returns
  (FinishApplicationMasterResponseProto);
  rpc allocate(AllocateRequestProto) returns(AllocateResponseProto);
}
```

同样，applicationclient_protocol. proto 文件定义了作业提交客户端 Client 与 ResourceManager 之间的协议 ApplicationClientProtocol；containermanagement_protocol. proto 文件定义了 ApplicationMaster 与 NodeManager 之间的协议 ContainerManagementProtocol；resourcemanger_administration_protocol. proto 文件定义了管理员端 Admin 与 ResourceManager 之间的通信协议 ResourceManagerAdministrationProtocol；ResourceTracker. proto 文件定义了 NodeManager 与 ResourceManager 之间的通信协议 ResourceTracker。另外，YARN 中的 RPC 协议参数也是由 Protocol Buffers 定义的（proto 文件），即 yarn_protos. proto 文件。该文件中定义了各个 RPC 协议的参数，用户可以根据实际业务需求来修改相应的配置参数（以 Hadoop 2.6.0 为例，各 PRC 协议可在 hadoop-yarn-project/hadoop-yarn/hadoop-yarn-api/src/main/proto/yarn_protos. proto 文件中进行配置）。

6.5 YARN 安全机制

YARN 的安全机制策略与前面所提到的 HDFS 的安全机制策略相同，包括认证（Authentication）和授权（Authorization）两部分。在 Hadoop 2.0 生态系统中，YARN 提供了两种认证机制和一种授权机制，下面将以 Hadoop 2.0 生态系统为例（以 Hadoop 2.6.0 stable 版本为例），分别对 YARN 所提供的授权机制和认证机制进行讲解。

6.5.1　认证机制

YARN 的认证机制与前面所介绍的 HDFS 的认证机制相同，都是采用 Kerberos 机制和 Simple 机制，如 Client 与 ResourceManager 之间初次通信采用 Kerberos 机制进行身份认证，而之后的通信便采用 Simple 机制来减小开销；NodeManager 与 ResourceManager 之间始终采用 Kerberos 认证机制（有关 Kerberos 认证机制的内容请查看 4.6.2 节内容）。要为 YARN 添加 Kerberos 认证机制，除需要修改 core-site.xml 中的 hadoop.security.authentication 的配置参数外，还需要生成 keytab 文件，并复制到其他的所有节点配置目录中，然后再修改 yarn-site.xml 中的相应配置参数，需要修改的配置参数如下所示。

```
<property>
  <description>The keytab for the resource manager.</description>
  <name>yarn.resourcemanager.keytab</name>
  <value>/etc/krb5.keytab</value>
</property>

<property>
  <description>The Kerberos principal for the resource manager.</description>
  <name>yarn.resourcemanager.principal</name>
  <value>yarn/_HOST@HADOOP.COM</value>
</property>

<property>
  <description>Keytab for NodeManager.</description>
  <name>yarn.nodemanager.keytab</name>
  <value>/etc/krb5.keytab</value>
</property>
<property>
  <description>The kerberos principal for the node manager.</description>
  <name>yarn.nodemanager.principal</name>
  <value>yarn/_HOST@HADOOP.COM</value>
</property>

<property>
  <description>who will execute(launch)the containers.</description>
  <name>yarn.nodemanager.container-executor.class</name>
<value>org.apache.hadoop.yarn.server.nodemanager.LinuxContainerExecutor
</value>
</property>
<property>
```

```
    <name>yarn.nodemanager.linux-container-executor.group</name>
    <value>yarn</value>
</property>
```

其中，yarn. nodemanager. container-executor. class 默认是 DefaultContainerExecutor，是以 NodeManager 的用户身份启动 Container 的，这里使用的 LinuxContainerExecutor，则是以提交 Application 的用户身份来启动和销毁 Container。

Simple 机制是 JAAS 协议与 Delegation Token 整合的一种认证机制，如在 Client 与 ResourceManager 之间初次通信时，并没有 Delegation Token 生成，所以第一次通信采用 Kerberos 机制进行身份认证。一旦通过认证，Client 将得到密钥（masterKey），并将 TokenIdentifier 和 masterKey 发送给 ResourceManager，ResourceManager 重新计算 TokenAuthenticator 和 Token，并检查该 Token 是否合法。如果 Token 是合法的，Client 和 ResourceManager 分别将 TokenAuthenticator 作为密钥、DIGEST-MD5 作为认证协议进行双方认证。

6.5.2 授权机制

YARN 的授权机制与 HDFS 相同，也是通过引入访问控制列表（Access Control List，ACL）实现的（有关 ACL 的授权机制的内容请查看 4.6.1 节内容）。要启动 YARN 的授权机制，要配置 $HADOOP_HOME/etc/Hadoop/core-site. xml 中的 hadoop. security. authorization 配置参数为 true，然后再对 $HADOOP_HOME/etc/Hadoop/hadoop-policy. xml 文件进行配置，该文件可以实现对 HDFS、MapReduce 和 YARN 的 ACL 配置，其中与 YARN 有关的配置项和协议之间的对应关系如表 6.2 所示。

表 6.2 配置项与协议之间的对应关系

配 置 项	协 议 名
security. resourcetracker. protocol. acl	ResourceTracker
security. resourcemanager-administration. protocol. acl	ResourceManagerAdministrationProtocol
security. applicationclient. protocol. acl	ApplicationClientProtocol
security. applicationmaster. protocol. acl	ApplicationMasterProtocol
security. containermanagement. protocol. acl	ContainerManagementProtocol
security. resourcelocalizer. protocol. acl	ResourceLocalizerProtocol
security. job. task. protocol. acl	TaskUmbilicalProtocol
security. job. client. protocol. acl	MRClientProtocol
security. applicationhistory. protocol. acl	ApplicationHistoryProtocol

其中，ResourceLocalizerProtocol 是 NodeManager 与 ResourceLocalizer 之间的通信协议；TaskUmbilicalProtocol 是 MapReduce Task 进程与后台父进程 TaskTracker 之间的通信协

议;MRClientProtocol 是 MapReduce JobClient 与 MapReduce ApplicationMaster 之间的通信协议;ApplicationHistoryProtocol 是 TimeLine Server 与 Generic History Service Client 之间的通信协议。

在本实例的配置文件 hadoop-policy.xml 中与 YARN 有关的默认配置内容如下。

```
<!-- YARN Protocols -->
<property>
  <name>security.resourcetracker.protocol.acl</name>
  <value> * </value>
</property>

<property>
  <name>security.resourcemanager-administration.protocol.acl</name>
  <value> * </value>
</property>

<property>
  <name>security.applicationclient.protocol.acl</name>
  <value> * </value>
</property>

<property>
  <name>security.applicationmaster.protocol.acl</name>
  <value> * </value>
</property>

<property>
  <name>security.containermanagement.protocol.acl</name>
  <value> * </value>
</property>

<property>
  <name>security.resourcelocalizer.protocol.acl</name>
  <value> * </value>
</property>

<property>
  <name>security.job.task.protocol.acl</name>
  <value> * </value>
</property>

<property>
  <name>security.job.client.protocol.acl</name>
```

```
    <value> * </value>
  </property>

  <property>
    <name>security.applicationhistory.protocol.acl</name>
    <value> * </value>
  </property>
```

其中，<value> * </value>表示所有用户都具有对应的服务操作权限。如果要实现不同用户或用户组具有不同的授权时，就需要根据实际需求进行配置(有关 YARN 不同权限的配置请查看 4.6.1 节内容)。

6.6 YARN 容错机制

前面介绍了 HDFS 通过数据块的多副本存储机制、NameNode 的单点失效解决机制、在 NameNode 和 DataNode 之间维持心跳检测、检测文件块的完整性、集群的负载均衡等多方面的方法来保证其容错能力。YARN 同样也要考虑 ApplicationMaster、NodeManager、ResourceManager 和 Container 组件的容错性，从而保证 Hadoop 集群系统的稳定运行。

1. ApplicationMaster 容错

前面提到，ApplicationMaster 主要负责应用程序的申请资源、监控任务运行和容错等。一旦 ApplicationMaster 运行超时或失败时，由 ResourceManager 负责重新启动 AppliationMaster。当 ApplicationMaster 重新启动后，就需要恢复之前的状态，而状态的恢复过程就是 ApplicationMaster 的容错。如 YARN 自带的 MapReduce 应用程序的 MRAppMaster，在作业执行过程中，MRAppMaster 会不断地将作业运行状态保存到 HDFS 上(哪些任务运行完成，哪些未完成等)，一旦 MRAppMaster 重启后，可通过保存在 HDFS 上的作业运行状态重新恢复原各个作业的状态，并只需要新运行未完成的那些任务即可。对于 ApplicationMaster 容错性的性能可通过 ApplicationMaster 的恢复性测试来确定，如测试 YARN 重启后作业恢复的速度来确定其 ApplicationMaster 的容错性。

2. NodeManager 容错

NodeManager 主要负责每个节点上的资源管理、任务执行、心跳上报等。如果 NodeManager 由于自身或网络因素在一定时间内没有向 ResourceManager 进行心跳上报，ResourceManager 将认为 NodeManager 及在上面正在运行的 Container 都已失效，并通知与其对应的 ApplicationMaster 重新分配资源等，由 ApplicationMaster 决定如何处理 NodeManager 中 Container 运行失败的任务。

3. ResourceManager 容错

由 YARN 的基本构架可以看出，ResourceManager 主要负责集群中所有资源的统一管理和分配，而且 ResourceManager 与 NameNode 相似，自身的容错性对 YARN 整体的容错性具有重大影响，也存在单节点故障。在 Hadoop 2.0 生态系统中，HDFS 实现了 NameNode 的 HA，同时在 Hadoop 2.4 版本以后，也开始实现了 ResourceManager 的 HA。ResourceManager 的 HA 与 NameNode 的 HA(请查看 4.7.3 节)基本原理相同，也是一种 Active/Standby 框架，在任意时刻，都有一个是 Active，其余处于 Standby 状态的 ResourceManager 可以随时转换成 Active 状态，该状态的转换可以通过在 CLI 输入命令手工完成，也可以自动完成。如基于 ZooKeeper 的 ResourceManager HA 就是一种可自动实现的切换技术，当 Active 状态的 ResourceManager 失效时，处于 Standby 状态的 ResourceManager 就会被选为 Active 状态，实现自己切换。有关 ResourceManager 的 HA 配置参数如表 6.3 所示。

表 6.3　ResourceManager HA 配置参数

配　置　项	描　　述
yarn.resourcemanager.zk-address	ZooKeeper 服务器地址
yarn.resourcemanager.ha.enabled	是否启用 ResourceManager HA，默认为 false
yarn.resourcemanager.ha.rm-ids	ResourceManager 的逻辑 ID 列表，用逗号分隔，如：rm1，rm2
yarn.resourcemanager.hostname.rm-id	每个 rm-id 的主机名，rm-id 的值包含在上面的参数值
yarn.resourcemanager.ha.id	用于标识 ResourceManager 的 ID 值
yarn.resourcemanager.ha.automatic-failover.enabled	是否启用自动故障转移。默认情况下，在启用 HA 时，启用自动故障转移
yarn.resourcemanager.ha.automatic-failover.embedded	启用内置的自动故障转移。默认情况下，在启用 HA 时，启用内置的自动故障转移
yarn.resourcemanager.cluster-id	集群的 ID，确保 ResourceManager 不会成为其他集群的 Active
yarn.client.failover-proxy-provider	由 Client、ApplicationMaster、NodeManager 用于故障恢复为 Active ResourceManager 时的类
yarn.client.failover-max-attempts	FailoverProxyProvider 尝试故障转移的最大次数
yarn.client.failover-sleep-max-ms	故障转移间的最大休眠时间(单位：ms)
yarn.client.failover-retries	每个尝试连接到 ResourceManager 的重试次数
yarn.client.failover-retries-on-socket-timeouts	当 Socket 超时时，每个尝试连接到 ResourceManager 的重试次数

有关 ResourceManager HA 的配置，可根据 4.7.3 节的 HDFS HA 作为参考(可将 Master1.Hadoop 的角色设为 rm1，即 Active ResourceManager；Master2.Hadoop 的角色设为 rm2，即 Standby ResourceManager)，实现 ResourceManager HA(查看状态的命令：yarn

rmadmin -getServiceState rm1；状态切换的命令：yarn rmadmin -transitionToStandby rm1）。有关 ResourceManager HA 的配置请查看本书配套资料中 Demo/YARN HA 文件夹中的相关内容。

4. Container 容错

Container 的容错机制是由 ApplicationMaster 决定的，当 ApplicationMaster 在一定时间内对已分配到的 Container 未及时启动，则 ApplicationMaster 会将该 Container 的状态设置为失效并回收；当 Container 在运行过程中，因为资源不足等因素导致运行失败，则由 ApplicationMaster 来决定如何处理该 Container。

6.7 YARN 资源调度机制

YARN 的资源调度机制是由 ResourceManager 中的 YarnScheduler 来负责完成的。在前面章节中有提到，YarnScheduler 是基于 Application 资源需求的分配策略，将资源分配给正在运行的 Application，如 CPU、内存、网络、磁盘等。在 Hadoop 1.0 生态系统中，YarnScheduler 默认的资源分配策略是 FIFO 实现的(FIFO Scheduler)，但还可以使用其他的资源调度机制，如 Fair Scheduler 和 Capacity Scheduler 等资源调度机制(有关这几种的资源调度机制的区别请查看表 6.1)。读者也可以根据自己的实际需求按照接口规范(ResourceScheduler)编写一个新的资源调度机制。

6.7.1 FIFO Scheduler

FIFO Scheduler 资源调度机制是 Hadoop 1.0 生态系统中默认的资源调度器(Hadoop 2.0 生态系统中的默认调度器已经改为 Capacity Scheduler)，它先按照作业的优先级高低，再按照到达时间的先后选择被执行的作业，其中，优先级越高分配的资源越多。该资源调度机制的优点是实现非常简单、调度过程快；缺点是对资源的利用率不高。

FIFO Scheduler 的资源调度器实际上是一个事件处理器，通过 ResourceManager 的异步事件调用机制处理来自外部的 9 种 Scheduler-EventType 类型的事件(以 Hadoop 2.6.0 版本为例)，并根据事件的具体含义进行相应的处理。调度器要处理的 9 种 Scheduler-EventType 类型的事件如表 6.4 所示。

表 6.4 调度器事件类型

处理类	事件	发送时机
Node	NODE_ADDED	当一个 NodeManager 被添加时
	NODE_REMOVED	当一个 NodeManager 被移除时
	NODE_UPDATE	当一个 NodeManager 跟 ResourceManager 进行心跳时
	NODE_RESOURCE_UPDATE	当一个 NodeManager 跟 ResourceManager 进行心跳时

续表

处 理 类	事　　件	发 送 时 机
RMApp	APP_ADDED	当一个新的应用被提交时
	APP_REMOVED	当一个应用被移除时
RMAppAttempt	APP_ATTEMPT_ADDED	当一个新的应用被提交时
	APP_ATTEMPT_REMOVED	当一个应用被移除时
ContainerAllocation-Expirer	CONTAINER_EXPIRED	当一个 Container 过期未被使用时

这 9 种类型的事件处理方式，请查看 hadoop-yarn-server-resourcemanager 项目工程下的 FifoScheduler 类中的 handle()方法，代码内容如下所示。

```
public void handle(SchedulerEvent event) {
    switch(event.getType()) {
    case NODE_ADDED:                    //增加总资源池的大小,修改状态等
    {
      NodeAddedSchedulerEvent nodeAddedEvent= (NodeAddedSchedulerEvent)event;
      addNode(nodeAddedEvent.getAddedRMNode());
      recoverContainersOnNode(nodeAddedEvent.getContainerReports(),
      nodeAddedEvent.getAddedRMNode());
    }
    break;
    case NODE_REMOVED:                  //删除一个 NodeManager,减少总资源池的大小
    {                                   //回收内存状态中的在该 NodeManager 上的 Container
      NodeRemovedSchedulerEvent nodeRemovedEvent= (NodeRemovedSchedulerEvent)event;
      removeNode(nodeRemovedEvent.getRemovedRMNode());
    }
    break;
    case NODE_RESOURCE_UPDATE:
    {                          //根据当前 NodeManager 状况,为某个 ApplicationMaster 分配 Container
      NodeResourceUpdateSchedulerEvent nodeResourceUpdatedEvent=
      (NodeResourceUpdateSchedulerEvent)event;
      updateNodeResource(nodeResourceUpdatedEvent.getRMNode(),
        nodeResourceUpdatedEvent.getResourceOption());
    }
    break;
    case NODE_UPDATE:
    {                          //根据当前 NodeManager 状况,为某个 ApplicationMaster 分配 Container
      NodeUpdateSchedulerEvent nodeUpdatedEvent=
      (NodeUpdateSchedulerEvent)event;
      nodeUpdate(nodeUpdatedEvent.getRMNode());
```

```
}
break;
case APP_ADDED:                          //如果接受应用,发送 APP_ACCEPTED 事件
{
  AppAddedSchedulerEvent appAddedEvent= (AppAddedSchedulerEvent)event;
  addApplication(appAddedEvent.getApplicationId(),
    appAddedEvent.getQueue(), appAddedEvent.getUser(),
    appAddedEvent.getIsAppRecovering());
}
break;
case APP_REMOVED:                        //清除内存中该应用的所有 Container
{                                        //对每个 Container 发送 KILL 事件
  AppRemovedSchedulerEvent appRemovedEvent= (AppRemovedSchedulerEvent)event;
  doneApplication(appRemovedEvent.getApplicationID(),
    appRemovedEvent.getFinalState());
}
break;
case APP_ATTEMPT_ADDED:                  //如果接受应用,发送 APP_ACCEPTED 事件
{
  AppAttemptAddedSchedulerEvent appAttemptAddedEvent=
  (AppAttemptAddedSchedulerEvent)event;
  addApplicationAttempt(appAttemptAddedEvent.getApplicationAttemptId(),
    appAttemptAddedEvent.getTransferStateFromPreviousAttempt(),
    appAttemptAddedEvent.getIsAttemptRecovering());
}
break;
case APP_ATTEMPT_REMOVED:                //清除内存中该应用的所有 Container
{                                        //对每个 Container 发送 KILL 事件
  AppAttemptRemovedSchedulerEvent appAttemptRemovedEvent=
  (AppAttemptRemovedSchedulerEvent)event;
  try {
    doneApplicationAttempt(
      appAttemptRemovedEvent.getApplicationAttemptID(),
      appAttemptRemovedEvent.getFinalAttemptState(),
      appAttemptRemovedEvent.getKeepContainersAcrossAppAttempts());
  } catch(IOException ie){
    LOG.error("Unable to remove application "
        + appAttemptRemovedEvent.getApplicationAttemptID(), ie);
  }
}
break;
case CONTAINER_EXPIRED:                  //修改内存中该 Container 相关的内存状态
```




```
        {
          ContainerExpiredSchedulerEvent containerExpiredEvent=
           (ContainerExpiredSchedulerEvent)event;
          ContainerId containerid= containerExpiredEvent.getContainerId();
          completedContainer(getRMContainer(containerid),
              SchedulerUtils.createAbnormalContainerStatus(
                  containerid,
                  SchedulerUtils.EXPIRED_CONTAINER),
              RMContainerEventType.EXPIRE);
        }
        break;
        default:
          LOG.error("Invalid eventtype "+ event.getType()+ ". Ignoring!");
        }
      }
```

在这里只能为读者抛砖引玉，具体的实现细节请读者认真阅读其相关源码，并结合有关资源调度机制的相关资料[72~77]，真正理解资源调度机制，最好能达到可以根据自身的业务需求，开发自己的资源调度器的能力。

6.7.2　Fair Scheduler

Fair Scheduler[72][73]资源调度机制是由 Facebook 开发的适合共享环境的调度器，Fair Scheduler 支持多队列，多用户共享资源，每个分组用户可以配置资源量，也可限制每个用户和每个分组中的并发运行作业数据。该资源调度机制的优点是支持作业分类调度，使不同类型的作业获得不同的资源分配，提高服务质量，动态调整并行作业数量，充分利用资源；缺点是不考虑节点的实际负载状态，导致节点负载实际不均衡。

Fair Scheduler 的资源调度机制同 FIFO Scheduler 资源调度机制相似，也需要处理 9 种不同 SchedulerEventType 类型的事件，Fair Scheduler 采用了三级资源分配策略，当一个节点上有空闲资源时，会依次选择队列、应用程序和 Container 使用该资源，具体的处理过程如下。

(1) Fair Scheduler 资源调度器收到资源申请后，先将这些请求存放到一个数据结构中，等待空闲资源出现后为其分配合适的资源。

(2) 当一个节点上有空闲资源时，首先需要选择队列。YARN 采用了层次结构组织队列，应用程序只存放在叶子队列，而其他非叶子队列主要用于计算叶子队列的资源量。Fair Scheduler 采用了深度优先遍历算法，从根队列开始，使用 FIFO、Fair 或 DRF (Dominant Resource Fairness)[77]策略对所有子队列进行排序。如果查找到子队列，则直接选择该队列，否则按上述方法继续查找叶子队列。

(3) 当选择好叶子队列之后，就需要选择应用程序，并按照 Fair 策略对叶子队列内部的应用程序进行排序，依次检查排序后的应用程序所需要的资源量与 Container 之间的匹配关系，从而为下一步 Container 的选择做准备。

（4）不同应用程序所请求的 Container 是不同的，如优先级、资源量等。Fair Scheduler 会根据优先级高低进行分配 Container，对于同一优先级的情况，会优先选择本地性的 Container。

如果要使用 Fair Scheduler 资源调度机制，需要修改两部分配置，即用于配置调度器相关参数的 yarn-site. xml 和自定义用于配置各个队列的资源量、权重等信息的配置文件 fair-scheduler. xml。如在 yarn-site. xml 文件中增加 yarn. resourcemanager. scheduler. class 配置项，内容如下。

```
<property>
    <description>The class to use as the resource scheduler.</description>
    <name>yarn.resourcemanager.scheduler.class</name>    <value>org.apache.hadoop.yarn.server.
resourcemanager.scheduler.fair.FairScheduler</value>
  </property>
```

然后需要在 ResourceManager 的配置目录下对 yarn-site. xml 文件增加相应的配置参数。表 6.5 给出了在 yarn-site. xml 文件中需要配置的相应参数及对应的描述。

表 6.5 yarn-site. xml 中 Fair Scheduler 配置参数

配 置 项	描 述
yarn. scheduler. fair. allocation. file	定义 XML 配置文件所在的位置
yarn. scheduler. fair. user-as-default-queue	默认为 true，表示指定用户名作为应用程序所在的队列名；如果设为 false，则表示所有未知队列的应用程序将被提交到 default 队列中
yarn. scheduler. fair. preemption	默认为 false，表示不支持资源抢占策略；如果设为 true，表示支持资源抢占
yarn. scheduler. fair. sizebasedweight	默认为 false，表示采用公平轮询的方式将资源分配给各个应用程序；如果设为 true，则表示按应用程序资源需求数目分配资源
yarn. scheduler. assignmultiple	默认为 false，表示不启用批量分配功能；如果设为 true，表示启用批量分配功能，即当一个节点出现大量资源时，可一次性分配完成
yarn. scheduler. fair. max. assign	默认为－1，表示不限制一次可分配的 Container 数量
yarn. scheduler. fair. locality. threshold. node	默认为－1，表示应用程序请求某个节点上的资源时，不跳过任何调度机会
yarn. scheduler. fair. locality. threshold. rack	表示当应用程序请求某个机架上的资源时，可跳过的最大资源调度机会
yarn. scheduler. increment-allocation-vcores	默认为 1，表示虚拟 CPU 规整化单位

同时需要在 ResourceManager 的配置目录下新建 fair-scheduler. xml 文件，并增加相应的配置参数。表 6.6 给出了 fair-scheduler. xml 文件中需要配置的相应参数及对应的描述。

表 6.6　fair-scheduler.xml 中 Fair Scheduler 配置参数

配　置　项	描　　述
minResources	最少资源保证量，设置格式为“X mb，Y vcores”
maxResources	最多可使用的资源量，即保证每个队列使用的资源量不超过该队列最多可使用的资源量
maxRunningApps	最多同时运行的应用程序数量
minSharePreemptionTimeout	最小共享量抢占时间，即一个资源池在该时间内使用的资源量低于最小资源量时，进行抢占资源
schedulingPolicy/schedulingMode	队列采用的调度模式，如 FIFO、Fair 或 DRF
aclSubmitApps	默认为“ * ”，表示任何用户均可向该队列提交应用程序
aclAdministerApps	队列的管理员列表，一个队列的管理员可管理队列中的资源和应用程序
maxRunningJobs	限制最多同时运行的应用程序数量
userMaxJobsDefault	用户的 maxRunningJobs 属性的默认值
defaultMinSharePreemptionTimeout	队列的 minSharePreemptionTimeout 属性的默认值
defaultPoolSchedulingMode	队列的 schedulingMode 属性的默认值
fairSharePreemptionTimeout	公平共享量抢占时间，即如果一个资源池在该时间内使用资源量低于公平共享量的一半，则开始抢占资源

有关 yarn-site.xml 和 fair-scheduler.xml 文件的配置请查看本书配套资料中 Demo/FairScheduler 文件夹中的相关内容。如在 Hadoop 2.6.0 版本中的 fair-scheduler.xml 文件中给出的相应配置内容如下所示。

```
<allocations>
  <user name="jenkins">
    <!--设置了三个队列 sls_queue_1、sls_queue_2、sls_queue_3,并且规定用户 jenkins 最多可同
    时运行 1000 个作业-->
    <maxRunningJobs>1000</maxRunningJobs>
  </user>
  <userMaxAppsDefault>1000</userMaxAppsDefault>
  <queue name="sls_queue_1">
    <minResources>1024 mb, 1 vcores</minResources>
    <schedulingMode>fair</schedulingMode>
    <weight>0.25</weight>
    <minSharePreemptionTimeout>2</minSharePreemptionTimeout>
  </queue>
  <queue name="sls_queue_2">
    <minResources>1024 mb, 1 vcores</minResources>
    <schedulingMode>fair</schedulingMode>
    <weight>0.25</weight>
```

```
      <minSharePreemptionTimeout>2</minSharePreemptionTimeout>
    </queue>
    <queue name="sls_queue_3">
      <minResources>1024 mb, 1 vcores</minResources>
      <weight>0.5</weight>
      <schedulingMode>fair</schedulingMode>
      <minSharePreemptionTimeout>2</minSharePreemptionTimeout>
    </queue>
  </allocations>
```

综上所述，Fair Scheduler 实现了资源的公平共享、基于任务数的负载均衡机制、调度策略的灵活配置(FIFO、Fair 或 DRF)，以及支持资源抢占的策略。如果读者需要更详细地了解 Fair Scheduler 资源调度机制及具体的实现，请查看 hadoop-yarn-server-resourcemanager 工程下 org. apache. hadoop. yarn. server. resourcemanager. scheduler. fair 包中的相关类和 org. apache. hadoop. yarn. server. resourcemanager. scheduler. fair. policies 包中的相关类，如图 6.6 所示。

```
◢ ⊞ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair
   > AllocationConfiguration.java
   > AllocationConfigurationException.java
   > AllocationFileLoaderService.java
   > FairScheduler.java
   > FairSchedulerConfiguration.java
   > FairSchedulerEventLog.java
   > FifoAppComparator.java
   > FSAppAttempt.java
   > FSLeafQueue.java
   > FSOpDurations.java
   > FSParentQueue.java
   > FSQueue.java
   > FSQueueMetrics.java
   > FSQueueType.java
   > FSSchedulerNode.java
   > MaxRunningAppsEnforcer.java
   > NewAppWeightBooster.java
   > QueueManager.java
   > QueuePlacementPolicy.java
   > QueuePlacementRule.java
   > Schedulable.java
   > SchedulingPolicy.java
   > WeightAdjuster.java
◢ ⊞ org.apache.hadoop.yarn.server.resourcemanager.scheduler.fair.policies
   > ComputeFairShares.java
   > DominantResourceFairnessPolicy.java
   > FairSharePolicy.java
   > FifoPolicy.java
```

图 6.6 Fair Scheduler 资源调度相关类

其中，AllocationConfiguration 和 AllocationConfigurationException 是配置文件相关类；FairScheduler 为资源调度器的主体部分；FairSchedulerConfiguration 用于配置常



量和默认值；FairSchedulerEventLog 用于把调度事件写到指定的 Log 文件中；QueueManager 用于维护一个组队列，并提供更新方法；SchedulingPolicy 用于不同的组内调度策略，如 FIFO、Fair、DRF 等；WeightAdjuster 是相应权重的修改接口；FifoPolicy 主要用于对提交作业的优先级排序等。

6.7.3 Capacity Scheduler

Capacity Scheduler[74]是由 Yahoo 开发的适合共享环境的资源调度器，支持多用户多队列管理，每个队列可以配置资源量，该资源调度器可以通过限制每个用户和每个队列的并发运行作业数据，以及限制每个作业使用的内存量等来防止同一个用户的作业独占队列中的资源。每个用户的作业有优先级，在单个队列中，作业按照 FIFO（实际上是先按照优先级，优先级相同的再按照作业提交时间）的原则进行调度，同时考虑用户资源量限制和内存限制。该资源调度机制的优点是支持多作业并行执行，提高资源利用率，动态调整资源分配，提高作业执行效率；缺点是用户需要了解大量系统信息，才能设置和选择队列。

Capacity Scheduler 的资源调度机制同 FIFO Scheduler 和 Fair Scheduler 资源调度机制相似，也需要处理 9 种不同 SchedulerEventType 类型的事件，Capacity Scheduler 同样也采用了三级资源分配策略，当一个节点上有空闲资源时，会依次选择队列、应用程序和 Container 使用该资源，具体的处理过程如下。

（1）Capacity Scheduler 资源调度器收到资源申请后，先将这些请求存放到一个数据结构中，等待空闲资源出现后为其分配合适的资源。

（2）当一个节点上有空闲资源时，首先需要选择队列。YARN 采用了层次结构组织队列，应用程序只存放在叶子队列，而其他非叶子队列主要用于计算叶子队列的资源量。Capacity Scheduler 与 Fair Scheduler 相同，都采用了深度优先遍历算法，从根队列开始，使用 FIFO、Fair 或 DRF 策略对所有子队列进行排序。如果查找到子队列，则直接选择该队列，否则按上述方法继续查找叶子队列。

（3）当选择好叶子队列之后，就需要选择应用程序，Capacity Scheduler 按照提交时间对叶子队列中的应用程序进行排序，依次检查排序后的应用程序所需要的资源量与 Container 之间的匹配关系，从而为下一步 Container 的选择做准备。

（4）不同应用程序所请求的 Container 是不同的，如优先级、资源量等。Capacity Scheduler 会根据优先级高低分配 Container，对于同一优先级的情况，会优先选择本地性的 Container。

虽然 Capacity Scheduler 和 Fair Scheduler 在实现过程中有很多相似处，都支持多用户多队列、单队列均支持优先级和 FIFO 调度方式、均支持资源共享等，但是其核心的调度策略不同。Capacity Scheduler 每次选择资源利用率低的 Queue，并同时考虑 FIFO 和内存制约因素，而 Fair Scheduler 调度策略是以提高资源利用率和减小集群管理为前提的，每次选择资源量需求最大的作业为调度对象。如果要选用 Capacity Scheduler 资源调度机制（在 Hadoop 2.0 生态系统中，默认为该资源调度机制），需要修改两部分配置，

即用于配置调度器相关参数的 yarn-site. xml 文件和自定义用于配置各个队列的资源量、权重等信息的配置文件 capacity-scheduler. xml。如在 yarn-site. xml 文件中增加 yarn. resourcemanager. scheduler. class 配置项，内容如下。

```
<property>
    <description>The class to use as the resource scheduler.</description>
    <name>yarn.resourcemanager.scheduler.class</name>    <value>org.apache.hadoop.yarn.server.
resourcemanager.scheduler.capacity.CapacityScheduler
</value>
  </property>
```

然后需要在 ResourceManager 的配置目录下对 yarn-site. xml 文件和 capacity-scheduler. xml 文件增加相应的配置参数。表 6.7 给出了 capacity-scheduler. xml 文件中需要配置的相应参数及对应的描述。

表 6.7 capacity-scheduler. xml 中 Capacity Scheduler 配置参数

配　置　项	描　　述
capacity	队列容量(百分比)，应保证每个队列的容量得到满足
maximum-capacity	队列最大容量(百分比)，队列使用共享资源时，不得超过其共享资源容量
minimum-user-limit-percent	每个用户最低资源保障(百分比)，即每个用户可使用资源量不能超过指定百分比
user-limit-factor	每个用户最多可使用的资源量(百分比)，即每个用户使用的资源量不能超过该队列容量的指定百分比
maximum-applications	默认值为 10 000，即集群或队列中同时处于等待和运行状态的应用程序数量的上限
root. queues	指定根队列
root. default. capacity	指定根队列的容量
node-locality-delay	表示当应用程序请求某个机架上的资源时，可跳过的最大资源调度机会
queue-mappings	用于指定作业队列映射列表
maximum-am-resource-percent	默认值为 0.1，即集群中用于运行应用程序 ApplicationMaster 的资源比例上限
resource-calculator	默认值为 DefaultResourceCalculator，即指定要使用的资源计算器为 DefaultResourceCalculator
state	用于设置队列状态，如 STOPPED 和 RUNNING。STOPPED 状态表示用户不可将应用程序提交到该队列及它的子队列；RUNNING 状态表示用户可以将应用程序提交到该队列及它的子队列
acl_submit_applications	限定哪些用户或用户组可向给定的队列提交应用程序
acl_administer_queue	为队列指定一个管理员，该管理员可控制该队列的所有应用程序

有关 yarn-site.xml 文件和 capacity-scheduler.xml 文件的配置请查看本书配套资料中的 Demo/CapacityScheduler 文件夹中的相关内容。如在 Hadoop 2.6.0 版本中的 capacity-scheduler.xml 文件中给出的相应配置内容如下所示。

```
<configuration>
  <property>
    <name>yarn.scheduler.capacity.maximum-applications</name>
    <value>10000</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.maximum-am-resource-percent</name>
    <value>0.1</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.resource-calculator</name>
   <value>org.apache.hadoop.yarn.util.resource.DefaultResourceCalculator
   </value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.queues</name>
    <value>default</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.default.capacity</name>
    <value>100</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.default.user-limit-factor</name>
    <value>1</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.default.maximum-capacity</name>
    <value>100</value>
  </property>

  <property>
    <name>yarn.scheduler.capacity.root.default.state</name>
```

```
        <value>RUNNING</value>
    </property>

    <property>
        <name>yarn.scheduler.capacity.root.default.acl_submit_applications
        </name>
        <value>*</value>
    </property>

    <property>
        <name>yarn.scheduler.capacity.root.default.acl_administer_queue</name>
        <value>*</value>
    </property>

    <property>
        <name>yarn.scheduler.capacity.node-locality-delay</name>
        <value>40</value>
    </property>

    <property>
        <name>yarn.scheduler.capacity.queue-mappings</name>
        <value></value>
    </property>

    <property>
        <name>yarn.scheduler.capacity.queue-mappings-override.enable</name>
        <value>false</value>
    </property>
</configuration>
```

综上所述,Capacity Scheduler 资源调度机制可通过设置资源最低保障和资源使用上限来进行容量保证,资源的灵活分配方式提高了资源利用率,通过 ACL 访问控制的方式提供了安全保证,以及在运行时就可随时调整分配参数的动态更新配置等。如果读者需要更详细地了解 Capacity Scheduler 资源调度机制,请查看 hadoop-yarn-server-resourcemanager 工程下 org. apache. hadoop. yarn. server. resourcemanager. scheduler. capacity 包中相关类,如图 6.7 所示。

其中,CapacitySchedulerConfiguration 是配置文件相关,用于配置常量和默认值;CapacityScheduler 为资源调度器的主体部分;CSQueue(Capacity Scheduler Queue)定义了一组树状节点应该提供的方法;LeafQueue(叶子队列)和 ParentQueue(父队列)是 CSQueue 的两个实现,用户提交的作业需要提交到叶子队列。

```
◢ ⊞ org.apache.hadoop.yarn.server.resourcemanager.scheduler.capacity
  > AbstractCSQueue.java
  > CapacityHeadroomProvider.java
  > CapacityScheduler.java
  > CapacitySchedulerConfiguration.java
  > CapacitySchedulerContext.java
  > CSAssignment.java
  > CSQueue.java
  > CSQueueUtils.java
  > LeafQueue.java
  > ParentQueue.java
  > PlanQueue.java
  > ReservationQueue.java
  > UserInfo.java
```

图 6.7 Capacity Scheduler 资源调度相关类

6.8 可在 YARN 上运行的框架

YARN 是 Hadoop 2.0 生态系统中的资源管理框架，不仅支持 MapReduce 的计算框架（默认计算框架），也可以作为其他计算框架的资源管理系统，如 Spark 实时计算框架、Storm 流式计算框架、Tez 有向图计算框架等。通常将运行在 YARN 上的计算框架称为“X On YARN”，如“Storm On YARN”，“Spark On YARN”，“Tez On YARN”等。除此之外，YARN 还支持其他类型的框架，如图 6.8 所示。

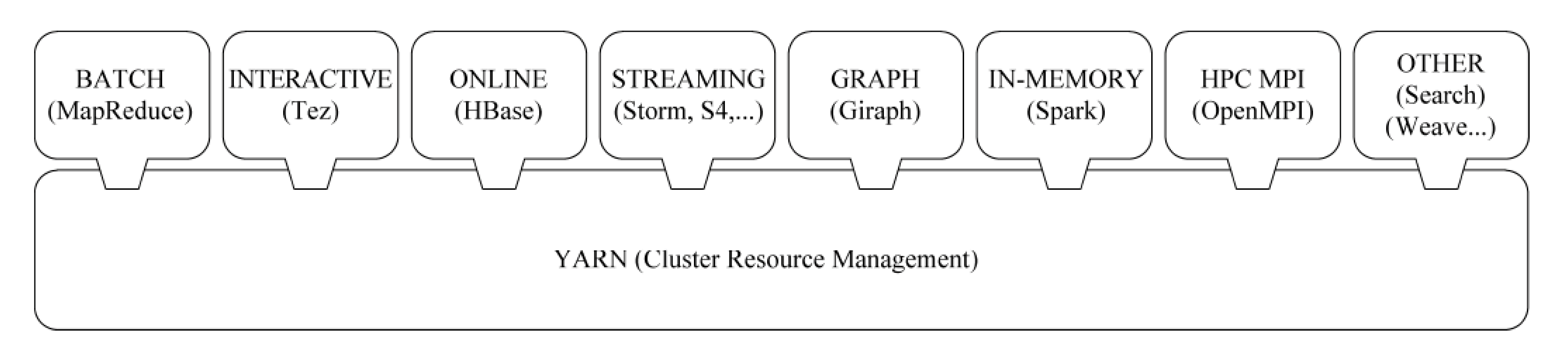


图 6.8 YARN 支持的框架

1. Storm on YARN

Storm 是一个免费、开源的分布式实时计算系统，与前面所介绍的 MapReduce 的批处理计算系统的区别在于 Storm 用于实时处理流式数据、简单易用、支持多种编程语言，目前 Storm 已升级成为 Apache 顶级项目。Storm 除了用于实时分析之外，还可用于在线机器学习、持续计算、分布式远程调用和 ETL 等领域。目前最新版本为 0.9.4，有关 Storm 更多的内容介绍请查看 Storm 官网（http://storm.apache.org/）了解。

2. Spark on YARN

Spark 是一个免费、开源的分布式实时计算系统，与前面所介绍的 MapReduce 和 Storm 的不同之处在于 Spark 启用了内存分布数据集，除了能够提供交互式查询外，还可

以优化迭代工作负载，目前 Spark 也已升级成为 Apache 顶级项目。目前最新版本为 1.3.1，有关 Spark 的更多内容请查看 Spark 官网(http://spark.apache.org/)了解。

3. Tez on YARN

Tez 是 Apache 最新开源的支持 DAG 作业的计算框架，其核心思想是将 MapReduce 计算框架的 Map 和 Reduce 两个阶段进一步拆分，将 Map 拆分为 Input、Processor、Sort、Merge 和 Output；Reduce 被拆分为 Input、Shuffle、Sort、Merge、Processor 和 Output 等。这些分解后的元操作可以进行灵活组合，从而形成一个大的 DAG 作业。Tez 计算框架在迭代计算和交互式计算方面比现有的 MapReduce 框架性能更好。目前最新版本为 0.6.0，有关 Tez 的更多内容请查看 Tez 官网(http://tez.apache.org/)了解。

4. S4 on YARN

S4[78](Simple Scalable Streaming System，分布式流计算系统)是由 Yahoo 发布的一个开源通用、分布式、可扩展、部分容错、具备可插拔功能的计算框架，该计算框架开发最初是用来处理用户的点击反馈，并用来解决搜索广告展现的实际问题。有关 S4 的更多内容请查看 S4 的开发者发表的一篇技术论文 *S4：Distributed Stream Computing Platform* 和 S4 官网了解。

5. Giraph on YARN

Giraph 是一个免费、开源的迭代的图计算框架，计算的输入是由点和直连的边组成的图，计算时将 MapReduce 中的 Mapper 进行封闭，而未使用 Reducer，并且在 Mapper 中进行多次迭代，每次迭代等价于 BSP 模型中的 SuperStep。Giraph 最早是由 Yahoo 开发并发布的，后来捐赠给 Apache 软件基金会。目前 Facebook 在开发图谱搜索服务时，选择采用了 Giraph 的计算框架，用来处理数万亿次用户及其行为之间的连接。有关 Giraph 的更多内容请查看 Giraph 的一篇技术论文 *Pregel：A System for Large-Scale Graph Processing*[79] 和 Giraph 官网了解。

6.9 YARN 编程实例

YARN 是一个资源管理框架，负责集群的资源管理和分配，如果要在 YARN 上运行非特定计算框架的应用程序，就需要实现自己的 Client 和 ApplicationMaster。本实例将以 Hadoop 2.6.0 版本中自带的 DistributedShell 应用程序为例，介绍如何将一个新的应用程序或新的计算框架运行在 YARN 之上。

6.9.1 编程过程

要将新的应用程序或计算框架运行在 YARN 之上，就需要编写 Client 和 ApplicationMaster 两个组件。如在 MapReduce on YARN 的编程实例中，YARN 已经为

MapReduce 实现了一个可直接使用的 Client 和 ApplicationMaster，即 MRClientService 和 MRAppMaster。

1. Client 过程

Client 端主要通过 RPC 通信协议与 ResourceManager 进行交互，负责向 ResourceManager 提交 ApplicationMaster，并查询应用程序的运行状态。Client 具体过程如下。

(1) Client 端首先需要通过 RPC 协议 ApplicationClientProtocol 中的 getNewApplication() 方法向 ResourceManager 提交应用程序请求 GetNewApplicationRequest，当 ResourceManager 收到 GetNewApplicationRequest 的请求后，返回该应用程序所需要的资源信息并进行应答 GetNewApplicationResponse。

(2) Client 将从 ResourceManager 上获取的应用程序 ID、名称、优先级、资源使用上下限等与应用程序相关的所有运行配置封闭到数据结构 ApplicationSubmissionContext 中，从而完成 ApplicationMaster 的初始化配置。

(3) Client 通过 ApplicationClientProtocol 中的 submitApplication() 方法将 ApplicationMaster 提交到 ResourceManager 上。

(4) ResouceManager 根据 ApplicationSubmissionContext 封装的应用程序内容及运行环境配置参数启动 ApplicationMaster。

(5) Client 通过多种方式查询应用程序的运行状态，如可通过 ApplicationMaster 或 ResourceManager 来获取应用程序的运行状态，并控制应用程序的运行过程。通常为了减少 ResourceManager 的压力，可使用从 ApplicationMaster 获取应用运行状态的方式。

如果读者想查看 Client 过程中涉及的有关类和相应内容，请查看 Hadoop API 或 hadoop-yarn-api 工程下 org. apache. hadoop. yarn. api 中的 ApplicationClientProtocol 类和 org. apache. hadoop. yarn. api. records 包中的 ApplicationSubmissionContext 类。

2. ApplicationMaster 过程

ApplicationMaster 主要负责向 ResourceManager 申请资源，并与 NodeManager 通信来启动各个 Container，而且还负责监控各个作业的运行状态和为失败的作业重新申请资源等。这就涉及 ApplicationMaster 与 ResourceManager 之间的通信协议 ApplicationMasterProtocol，ApplicationMaster 与 NodeManager 之间的通信协议 ContainerManagementProtocol。ApplicationMaster 具体过程如下。

(1) Client 向 ResouceManager 提交应用程序后，ResourceManager 会根据提交的信息来分配一定资源并启动 ApplicationMaster。ApplicationMaster 会通过 ApplicationMasterProtocol 协议中的 registerApplicationMaster()方法主动向 ResourceManager 发送注册请求，进行注册。

(2) ApplicationMaster 完成注册后，再通过 ApplicationMasterProtocol 协议中的 allocate() 方法向 ResourceManager 申请运行应用程序所需要的资源。ApplicationMasterProtocol 中的 allocate() 方法不仅具有向 ResourceManager 申请资源的功能，还具有向

ResourceManager 归还资源，以及周期性地在 ApplicationMaster 与 ResourceManager 之间发送心跳的功能。

（3）ApplicationMaster 获取资源后，通过 ContainerManagementProtocol 的 startContainers()方法在 NodeManager 上启动 Containers。

（4）ApplicationMaster 不断重复（2）和（3），直到所有应用程序运行完成后，ApplicationMaster 通过 ApplicationMasterProtocol 协议中的 finishApplicationMaster()方法向 ResourceManager 汇报应用程序的最终状态，并注销 ApplicationMaster。

如果读者想查看 ApplicationMaster 过程中涉及的有关类和相应内容，请查看 Hadoop API 或 hadoop-yarn-api 工程下 org. apache. hadoop. yarn. api 中的 ApplicationMasterProtocol 类和 ContainerManagermentProtocol 类。

6.9.2 DistributedShell 实例

DistributedShell 应用程序是 YARN 自带的一个非常简单的应用程序实现，可看作 YARN 编程中的"Hello World"，其主要功能是并行执行用户提供的 Shell 命令或者 Shell 脚本。本节将通过 DistributedShell 应用程序实例来介绍如何进行 YARN 编程。

1. 编写 Client

DistributedShell 应用程序是由 Client 类来完成其 Client 功能的，Client 的入口为 main 函数，main 函数的代码内容如下。

```
public static void main(String[] args){
        Client client=new Client();
        …
          boolean doRun=client.init(args);
         …
        result=client.run();
         …
   }
```

DistributedShell 应用程序中 Client 中最重要的函数为 run()，该函数实现了 Client 的功能，其代码内容如下。

```
public boolean run()throws IOException, YarnException {
      yarnClient.start();
      //与 ResourceManager 通信,获得 Application ID
      YarnClientApplication app=yarnClient.createApplication();
      GetNewApplicationResponse appResponse=app.getNewApplicationResponse();
      …
      //ApplicationMaster 的初始化配置
      ApplicationSubmissionContext appContext=
      app.getApplicationSubmissionContext();
      ApplicationId appId=appContext.getApplicationId();
```

```
    …
    //创建一个用于运行 ApplicationMaster 的 Container
    //Container 相关信息被封装到 ContainerLaunchContext 对象中
    ContainerLaunchContext amContainer = ContainerLaunchContext.newInstance (localResources, env,
    commands, null, null, null);

    //配置 ApplicationMaster 所需的资源
    Resource capability=Resource.newInstance(amMemory, amVCores);
    appContext.setResource(capability);
    appContext.setAMContainerSpec(amContainer);

    //设置 ApplicationMaster 的优先级
    Priority pri=Priority.newInstance(amPriority);
    appContext.setPriority(pri);
    …
    //提交 ApplicationMaster 到 ResourceManager,从而完成作业提交功能
    yarnClient.submitApplication(appContext);
    //监控应用程序的运行状态
    return monitorApplication(appId);
  }
```

关于 DistributedShell 应用程序 Client 的具体内容,请查看本书配套资料中 DistributedShell 工程下的 Client 类。

2. 编写 ApplicationMaster

DistributedShell 应用程序是由 ApplicationMaster 类来完成其 ApplicationMaster 功能的,ApplicationMaster 类的入口为 main 函数,main 函数的代码内容如下。

```
public static void main(String[] args){
    …
      ApplicationMaster appMaster=new ApplicationMaster();
      //初始化 ApplicationMaster
      boolean doRun=appMaster.init(args);
      //运行 ApplicationMaster
      appMaster.run();
      //向 ResourceManager 归还资源
      result=appMaster.finish();
      …
    }
```

DistributedShell 应用程序中 ApplicationMaster 最重要的函数为 run(),该函数实现了 ApplicationMaster 的功能,其代码内容如下。

```
public void run()throws YarnException, IOException {
…
    //ApplicationMaster 与 ResourceManager 通信,注册自己
    //开始发送心跳到 ResourceManager
    RegisterApplicationMasterResponse response=amRMClient
      .registerApplicationMaster(appMasterHostname, appMasterRpcPort,
      appMasterTrackingUrl);
    …
    //向 ResourceManager 申请资源
    //启动 Containers
    for(int i=0; i<numTotalContainersToRequest;++i){
      ContainerRequest containerAsk=setupContainerAskForRM();
      amRMClient.addContainerRequest(containerAsk);
    }
    …
  }
```

关于 DistributedShell 应用程序 ApplicationMaster 的具体实现细节,请查看本书配套资料中 DistributedShell 工程下的 ApplicationMaster 类。

3. 打包运行

当写好 Client 和 ApplicationMaster 之后,就可以对该 YARN 应用程序进行测试、打包和运行,具体的测试和打包方法请查看 5.6 节的 MapReduce 开发实例的相关内容。这里已对 DistributedShell 实例进行测试和打包,下一步将通过输入命令行来对 DistributedShell 应用程序进行运行。DistributedShell 的基本运行参数如下所示。

```
-appname <arg>                  应用程序名
-container_memory <arg>         表明每个 Container 需要的内存资源
-container_vcores <arg>         表明每个 Container 需要的虚拟内核数
-help                           输出帮助信息
-jar <arg>                      表明运行 ApplicationMaster 的应用程序
-master_memory <arg>            表明 ApplicationMaster 需要的内存资源
-master_vcores <arg>            表明 ApplicationMaster 需要的虚拟内核数
-priority <arg>                 表明优先级
-shell_command <arg>            表明每个 Container 上执行的真正应用程序
-shell_script <arg>             表明每个 Container 上执行的真正应用程序
…
```

本实例的输入参数如下。

```
[hadoop@master1 ~]$ hadoop jar
/hadoop/yarn/hadoop-yarn-applications-distributedshell-2.6.0.jar
```

```
org.apache.hadoop.yarn.applications.distributedshell.Client -jar /
hadoop/yarn/hadoop-yarn-applications-distributedshell-2.6.0.jar
-shell_command ls\ -shell_script ignore.sh\ -num_containers 5\
-container_memory 200\ -master_memory 200\ -priority 10
```

DistributedShell 的运行日志如下。

```
15/04/23 22:44:49 INFO distributedshell.Client: Initializing Client
15/04/23 22:44:49 INFO distributedshell.Client: Running Client
15/04/23 22:44:49 INFO client.RMProxy: Connecting to ResourceManager at master1.hadoop/192.168.85.100:
8032
15/04/23 22:44:52 INFO distributedshell.Client: Got Cluster metric info from ASM, numNodeManagers=3
15/04/23 22:44:52 INFO distributedshell.Client: Got Cluster node info from ASM
…
15/04/23 22:45:05 INFO distributedshell.Client: Application has completed successfully. Breaking
monitoring loop
15/04/23 22:45:05 INFO distributedshell.Client: Application completed successfully
```

到此,YARN 的 DistributedShell 实例的基本过程已经解析完成,请读者通过该实例深入理解 YARN 的编程过程,达到触类旁通为止。

第7章 分布式列存储数据库 HBase

分布式列存储数据库 HBase 是一个高可靠性、高性能、面向列、可伸缩的分布式存储系统，类似于 Google BigTable 利用 GFS 的文件存储系统。本章将介绍 HBase 的基本特点、体系架构、安装配置、数据模型、关键技术及交互接口等内容。

7.1 HBase 概述

HBase 是一个分布式的、面向列的开源 NoSQL 数据库，能够对大型数据提供随机、实时的读写访问，是基于 Google 的 BigTable 的开源实现[36]。Google 的 BigTable 利用 GFS 作为其文件存储系统，HBase 利用 Hadoop 的 HDFS 作为其文件存储系统；Google 通过 MapReduce 来处理 BigTable 中的海量数据，HBase 同样利用 Hadoop 中的 MapReduce 来处理 HBase 中的海量数据；Google BigTable 利用 Chubby 作为协同服务，HBase 则利用 ZooKeeper 作为协同服务。HBase 在 Hadoop 之上提供了类似于 BigTable 的能力，主要用于存储并处理大规模的数据，与传统关系型数据库的不同之处在于，它是一个适合于结构化、半结构化和非结构化数据存储的数据库；不是基于行的存储模式，而是基于列的存储模式。

随着 Hadoop 的良好发展和 2006 年 Google 发表的 *A Distributed Storage System for Structured Data*[36]论文为 HBase 的诞生提供了一些基础性的理论和实践基础。2007 年 4 月，Cafarella 发布了实验性的、基本可用的 BigTable 系统代码，并称其为 HBase，然后由 Powerset 公司的 Jim Kellem 接手继续推进该项目。2007 年 10 月，HBase 的第一个发布版本是由 Hadoop 0.15.0 捆绑在一起发布的。HBase 发展简史如表 7.1 所示。

表 7.1 HBase 发展历程

时　间	事　件
2006 年 11 月	Google 公司 Chang 等人发表的 BigTable 论文
2007 年 4 月	基于 BigTable 的 HBase 源码基本可用
2007 年 10 月	HBase 第一个发布版本(Hadoop 0.15.0)
2008 年 1 月	Hadoop 成为 Apache 的顶级项目，HBase 成为 Hadoop 的子项目

续表

时　间	事　件
2008 年 10 月	HBase 0.18.1 发布
2009 年 1 月	HBase 0.19.0 发布
2009 年 9 月	HBase 0.20.0 发布
2010 年 5 月	HBase 成为 Apache 的顶级项目
2010 年 6 月	HBase 0.89.20100621，第一个开发版本
2011 年 1 月	HBase 0.90.0 发布，稳定性和持久性有所提升
2012 年 1 月	HBase 0.92.0 发布，支持协处理器和安全控制
2012 年 5 月	HBase 0.94.0 发布，包含很多性能的提升和 Bug 修复及新特性
2013 年 4 月	HBase 0.95.0 发布，性能提升、Bug 修复及新特性
2013 年 10 月	HBase 0.96.0 发布，提高了稳定性、平均恢复时间(MTTR)等
2014 年 2 月	HBase 0.98.0 发布，提高了性能和安全特性
2014 年 11 月	HBase 0.99.0 发布，提高了性能和安全特性
2015 年 2 月	HBase 1.0.0 发布，简化和提升 Region Assignment 的可靠性等
2015 年 4 月	HBase 2.0.0-SNAPSHOT 发布，更好地支持 Cell 级别的 ACL 控制等

HBase 的主版本与 Hadoop 的主版本号是相互匹配的，如 HBase 0.19.x 就表示它是工作在 Hadoop 0.19.X 之上的。因此，不同的 Hadoop 版本需要选择不同的 HBase 版本。Hadoop 版本与 HBase 版本之间的对应关系如表 7.2 所示。

表 7.2　Hadoop 版本与 HBase 版本之间的对应关系

	HBase 0.92.x	HBase 0.94.x	HBase 0.96.x	HBase 0.98.x	HBase 1.0.x
Hadoop 0.20.x	支持并测试	不支持	不支持	不支持	不支持
Hadoop 0.22.x	支持并测试	不支持	不支持	不支持	不支持
Hadoop 1.0.0-1.0.2	不支持	不支持	不支持	不支持	不支持
Hadoop 1.0.3+	支持并测试	支持并测试	支持并测试	不支持	不支持
Hadoop 1.1.x	支持未测试	支持并测试	支持并测试	不支持	不支持
Hadoop 0.23.x	不支持	支持并测试	支持未测试	不支持	不支持
Hadoop 2.0.x	不支持	支持未测试	不支持	不支持	不支持
Hadoop 2.1.0	不支持	支持未测试	支持并测试	不支持	不支持
Hadoop 2.2.0	不支持	支持未测试	支持并测试	支持并测试	支持未测试
Hadoop 2.3.x	不支持	支持未测试	支持并测试	支持并测试	支持未测试
Hadoop 2.4.x	不支持	支持未测试	支持并测试	支持并测试	支持并测试
Hadoop 2.5.x	不支持	支持未测试	支持并测试	支持并测试	支持并测试
Hadoop 2.6.x	不支持	支持未测试	支持并测试	支持并测试	支持并测试

从表 7.2 可以看出，随着 HBase 版本的发展，为了满足多 Hadoop 版本的支持，一个 HBase 版本可以工作于多个 Hadoop 版本之上，不再只工作于与其主版本匹配的某一个 Hadoop 版本中。从 HBase 0.90.0 主版本开始，版本号的命名遵循了奇偶发布版本号的惯例，即奇数版本号为开发人员预览版(Developer Previews)，偶数版本号为可以产品化的稳定版(Stable)。因此，稳定版的发布版本包括 0.90、0.92、0.94、0.96、0.98、1.0.0 等。

7.2 HBase 特点

传统的数据库系统已无法适应大型分布式数据存储的需要，而且关系模型对数据的操作使数据的存储变得异常复杂，从而需要一种适用于存储不同种类的数据源、不同类型数据格式、不强调数据之间的关系，并且满足大规模数据处理的数据库。HBase 就是这样一个数据库，它是一种高可靠性、高性能、面向列、可伸缩、实时读写的分布式存储系统，利用 HBase 技术可实现大规模的数据存储集群。HBase 从设计理念上就为可扩展做好了充分准备，存储空间的扩展只需要加入存储节点即可线性扩展，使用不同于关系型数据库中“表”的概念(实质上是一张极大的、非常稀疏的、不支持 SQL、存储在分布式文件系统上的表)的实时读写的分布式文件存储系统，其具体特点如下。

1. 强一致性

HBase 在设计时选择了分布式系统 CAP 理论中的 CP，并保证单行的 ACID(原子性 Atomicity、一致性 Consistency、隔离性 Isolation 和持久性 Durability)，即 HBase 是强一致性的。其中，CAP 是 Consistency(一致性)、Availability(可用性)和 Partition-tolerance(分区容错)的缩写。Consistency(一致性)是指在分布式系统中的所有数据备份，在同一时刻是否具有同样的值；Availability(可用性)是指在集群中一部分节点故障后，集群整体是否还能响应客户端的读写请求；Partition-tolerance(分区容错)是指集群中的某些节点在无法联系后，集群整体是否还能继续进行服务。HBase 的强一致性主要体现在每个 HRegion 同时只有一台 HRegionServer 为它服务，对一个 HRegion 所有的操作请求，都由这台 HRegionServer 来响应。

2. 行事务

HBase 提供行级的事务，并且同一行的列的写入是原子操作。在 HBase 0.94 版本之前，HBase 支持的事务是很有限的，每次行事务只能执行一个写操作，比如连续地执行一系列 Put 或 Delete 操作，这些操作是单独一个个的事务，其整体并不是原子性执行的。在 HBase 0.94 版本之后，HBase 具备更完整的事务支持，可以实现 Put 或 Delete 在同一个事务中一起原子性执行(具体请查看 HBASE-3584，https://issues.apache.org/jira/browse/HBASE-3584)。

3. 实时读写

HBase 通过行锁和 MVCC(Multiversion Concurrency Control,多版本并发控制技术)保证了高效的并发读写。建议读者使用 HBase 系统自身提供的性能测试工具(./bin/hbase org.apache.hadoop.hbase.PerformanceEvaluation)对 HBase 的随机读写进行性能测试。HBase 自己带的性能测试工具提供了随机读写、多客户端读写等性能测试功能。作者根据工具测试的结果看,HBase 的实时读写性能不算差。

4. 面向列

HBase 是面向列(Column-Oriented)的存储机制,与 RDBMS 的面向行(Row-Oriented)的存储机制不同,在面向列的存储机制下对于 Null 值的存储是不占用任何空间的,如在 HBase 中某个表有 10 列,但在存储时只有一列有数据,那么无数据的 9 列就不占用存储空间,而对于 RDBMS 的面向行存储时,则 10 列全部占用存储空间。因此,HBase 更适合存储稀疏的半结构化或非结构化的数据。

5. 可伸缩性

当硬件出现故障时,HBase 能够轻易地增加或者减少硬件数量,而且对错误的兼容性比较高。HBase 的可伸缩性还体现在 HRegion 的自动分裂;Master 的 Balance 策略保证 HRegion 数的均衡;通过增加 DataNode 即可增加容量;通过增加 HRegionServer 即可增加读写吞吐量等。

7.3 HBase 体系架构

HBase 的体系架构也是一个典型的 Master/Slave 架构,主要包括 HBase Client、ZooKeeper、HMaster、HRegionServer 和 Store、HLog 等,其体系架构如图 7.1 所示。

从图 7.1 可以看出,用户可通过 Client 读写数据;HMaster 负责管理元数据(表分区,管理该分区的 HRegionServer);HRegionServer 负责 HRegion 数据的存储和读取;ZooKeeper 主要负责协调 HBase 中的所有 HRegionServer,并处理 HBase 运行期间可能遇到的错误;HBase 逻辑上的表可能会被划分成多个 HRegion,然后将 HBase 的所有数据(HLog 和 HFile)均存储到 HDFS 上;HDFS 将文件划分成 64MB 的 Block,并存储多个副本。HBase 体系架构中各组件的具体功能描述如下。

1. HBase Client

HBase Client 主要用于提供访问 HBase 的接口,并维护着一些 Cache 来加快对 HBase 的访问,如 Client 通过 RPC 通信机制与 HMaster 通信进行管理类操作;Client 通过 RPC 机制与 HRegionServer 通信进行数据读写类操作。

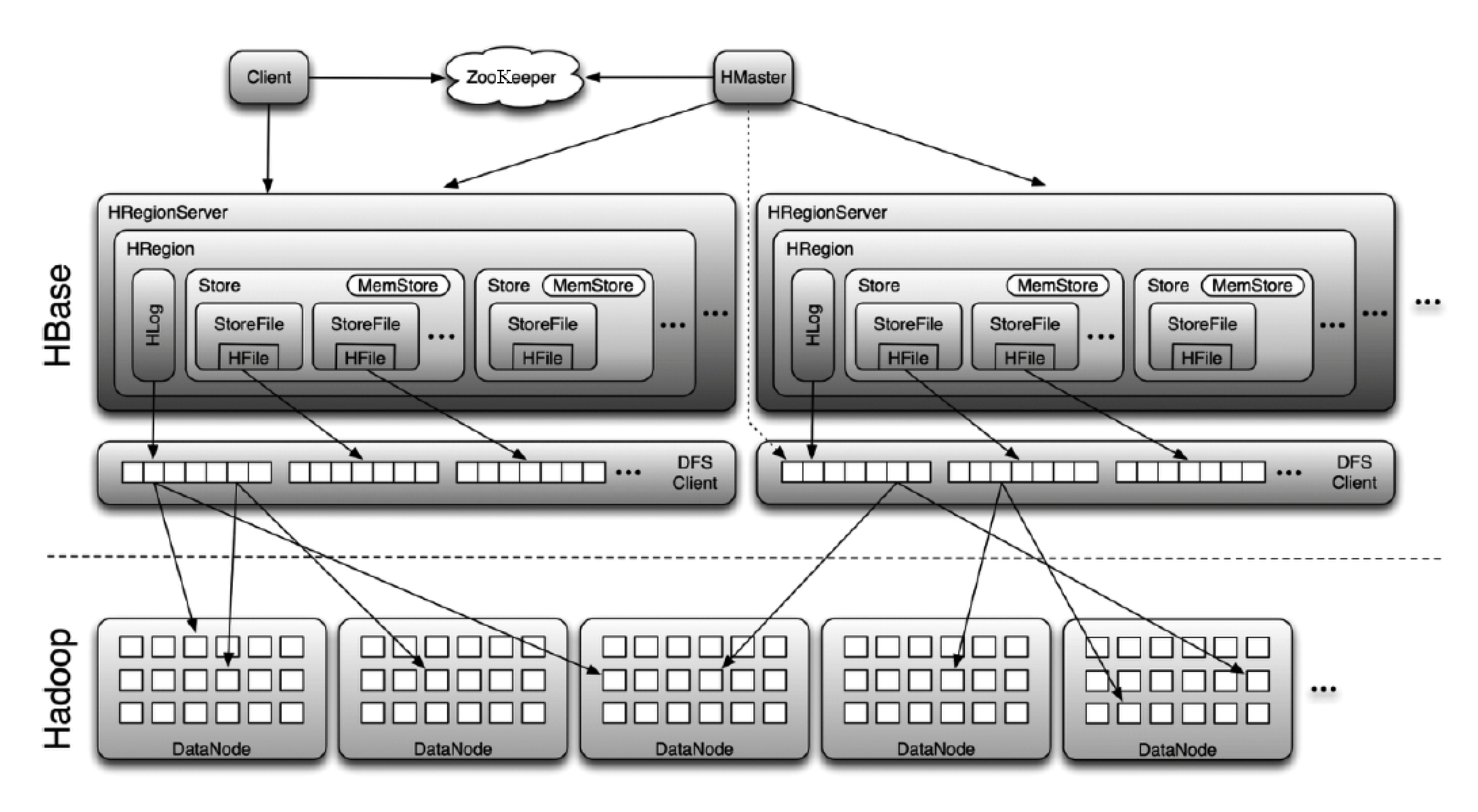


图 7.1　HBase 体系架构

（图片来源：http://www.toadworld.com/）

2. ZooKeeper

ZooKeeper 为 HBase 提供了稳定服务和 failover 机制，通过 ZooKeeper 的 Master Election 机制解决 HMaster 的单节点故障。ZooKeeper Quorum 不仅存储了-ROOT-表的地址(HBase 中有两张特殊的 Table 表，-ROOT-和.META.，.META 表可以有多个 HRegoin，记录了用户表的 Region 信息；-ROOT-表只允许有一个 HRegion，记录了.META表的 HRegion 信息)，而且也存储了 HMaster 的地址，保证任何时候集群中只有一个 HMaster。ZooKeeper 还实时监控 HRegionServer 的状态，并将 HRegionServer 的相关信息实时通知给 HMaster。

3. HMaster

HMaster 主要负责 Table 表和 HRegion 的管理工作，如管理用户对 Table 表的增、删、查、改操作；管理 HRegionServer 的负载均衡，并调整 HRegion 分布；当 HRegion 分裂后，负责重新分配 HRegion；当 HRegionServer 停机后，负责对失效 HRegionServer 上的 HRegions 迁移等。HMaster 还负责监控每个 HRegionServer 的健康状况，如果某个 HRegionServer 不可用，HMaster 将会由其他 HRegionServer 来代替该 HRegionServer 为其提供服务。

4. HRegionServer

HRegionServer 主要负责响应用户的 I/O 请求，并向 HDFS 文件系统中读写数据，是 HBase 中最核心的模块。HRegionServer 还负责与 HMaster 之间进行通信，获取自己

所需要服务的数据表，并向 HMaster 反馈自己的运行状况。

5. Store

Store 存储是 HBase 的存储核心，每个 Store 又由一个 MemStore 和零个或多个 StoreFiles 组成。其中，MemStore 是 Stored Memory Buffer，用户写入的数据首先会放入 MemStore 中，当 MemStore 满后，会 Flush 成一个 StoreFile（底层实现为 HFile）。当 StoreFile 文件数量增长到一定阈值时，就会通过 Compact 合并操作，将多个 StoreFiles 通过版本合并和数据删除等操作合并成一个 StoreFile，从而保证了 HBase 的 I/O 性能。

6. MemStore

MemStore 用于存储修改的数据 KeyValues，MemStore 是放在内存中的。当 MemStore 的大小达到一个阈值（默认为 64MB）时，MemStore 会被 HBase 进行 flush 操作，从而生成一个快照。

7. StoreFile

MemStore 内存中的数据写到文件后就是 StoreFile，StoreFile 的底层实现是以 HFile 的格式保存在 HDFS 上的。

8. HRegion

HRegion 是实际存储数据的区域，是分布式存储和负载均衡的最小单元（不是存储的最小单元，存储的最小单元为 HFile），它包含一个或多个 Store。最小单元就表示不同的 HRegion 可以分布在不同的 HRegionServer 上，但一个 HRegion 不会拆分到多个 HRegionServer 上。

9. HLog

HLog 主要用于保存用户操作 HBase 的日志信息，其存储格式为 WAL（Write Ahead Log），物理上是 Hadoop 的 Sequence File 文件。用户的所有操作都会先记录到 HLog 中，然后再保存到 HRegion 中。由于 MemStore 驻留在内存中，保存数据时会先存储到 MemStore 中，然后再根据用户设定的显式或隐式的刷写模式将数据保存到 HFile 中。当出现数据已保存到 MemStore，还没有刷写到 HFile 而出现异常时，HLog 就会将用户操作的指令保存起来，并重新执行这些指令，完成数据的存储。

10. HFile

HFile 主要负责实际数据的存储，是 Hadoop 的二进制格式文件，也是 HBase 中存储的最小单元。用户可根据业务需求对 HFile 进行重新拆分，让数据更加分散，从而提高数据读取效率。

7.4 HBase 安装配置

要使得 HBase 在 Hadoop 集群中充分发挥作用，就需要相应的软件、硬件及以太网络的支撑。下面将针对 HBase 系统正常运行所需要的软硬件环境、网络环境、安装选项、运行模式、配置、部署等方面进行介绍，帮助初学者对 HBase 进行更深入的研究和学习。

7.4.1 准备工作

一个完整的 Hadoop 集群在安装前需要做好 Hadoop 集群的规划，即软硬件环境、网络环境、各节点上安装的操作系统、集群参数配置等一系列过程。HBase 是 Hadoop 生态系统中的一个组件，因此，在 Hadoop 集群规划时，就应该考虑到对 HBase 的集群规划（有关集群规划的相关内容请查看 3.6 节）。在没有实际业务需求，以研究和学习为目的，并帮助初学者快速搭建 HBase 集群环境的情况下，本实例将在表 3.7 的 Hadoop 集群搭建的基础上（软硬件环境及网络环境）对 HBase 集群中的各节点角色进行分配，具体节点角色分配和配置参数信息如表 7.3 所示。

表 7.3 HBase 集群节点角色分配

机器名称	角　　色	IP 地址	硬件参数	操作系统
Master1. Hadoop	NameNode SecondaryNameNode ResourceManager HMaster	192. 168. 1. 100	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave1. Hadoop	DataNode NodeManager HRegionServer ZooKeeper	192. 168. 1. 101	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave2. Hadoop	DataNode NodeManager HRegionServer ZooKeeper	192. 168. 1. 102	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave3. Hadoop	DataNode Nodemanager HRegionServer ZooKeeper	192. 168. 1. 103	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7

由表 7.3 可以看出，Master 机器主要配置 HMaster 角色，主要负责 Table 和 Region 的管理工作；Slave 机器配置 HRegionServer 和 ZooKeeper 角色，主要负责响应用户 I/O 请求、向 HDFS 文件系统中读写数据、HRegionServer 的状态监控等。

HBase 的软硬件环境、网络环境、各节点操作系统等都规划好之后，还要选择 HBase 所使用的底层分布式文件系统。因为 HBase 本身无法复制和维护自身存储文件的副本，需要底层的文件存储系统保证其数据存储的可靠性、容错性和可扩展性。在 Hadoop 生

态系统中，HDFS 成为 HBase 最常使用的文件系统作为底层存储，当然也可以使用其他任何支持 Hadoop 接口的文件系统代替 HDFS，如本地文件系统、S3 等。本实例 HBase 选择默认的 Hadoop 文件系统 HDFS，该文件系统是在生产实践中广泛使用并经过检验的分布式文件系统。

7.4.2　安装 HBase

HBase 的安装方式主要有两种，一种是源码编译型的安装方式，另一种是二进制发布包的安装方式。源码编译型的安装方式需要安装 Maven 以及完整的 Java 开发工具包(Java Development Kit，JDK)来准备 HBase 的编译和运行环境，再对 HBase 源码进行编译和打包，从而完成 HBase 的安装。本实例将采用第二种 HBase 的安装方式，即采用二进制发布包的安装方式。首先需要从 HBase 官网(http://hbase.apache.org/)下载 HBase 的最新版本(到目前为止 HBase 最新版为 2.0.0-SNAPSHOT，本实例下载了 HBase-1.0.0 版本，该版本的 HBase 本身已带 ZooKeeper，不需要单独安装)，并将内容解压到适当目录中，具体命令如下。

```
#tar -zxvf hbase-1.0.0                  --解压 HBase二进制文件
#mv hbase-1.0.0 /opt/                   --移动 hbase-1.0.0到"/opt/"目录下
                                        --把"/opt/hbase-1.0.0"读权限分配给 hadoop用户
#chown -R hadoop:hadoop /opt/hbase-1.0.0
#vi /etc/profile                        --把 HBase安装路径添加到配置文件 profile中
  export HBASE_HOME=/opt/hbase-1.0.0
  export PATH=$PATH:$HBASE_HOME/bin
#su hadoop                              --切换为 hadoop用户
```

通过上述命令，就完成了 HBase 二进制文件的解压、hadoop 权限、配置环境变量等过程，将所有的 Master 节点和 Slave 节点都进行上述操作后，HBase 系统就安装成功了。安装成功后 HBase 的文件目录如图 7.2 所示。

```
drwxr-xr-x.  7 hadoop hadoop   4096 May  4 04:52 .
drwx------. 24 hadoop hadoop   4096 May  4 04:56 ..
drwxr-xr-x.  4 hadoop hadoop   4096 May  4 04:52 bin
-rw-r--r--.  1 hadoop hadoop 130672 Feb 14 22:40 CHANGES.txt
drwxr-xr-x.  2 hadoop hadoop   4096 May  4 04:52 conf
drwxr-xr-x. 12 hadoop hadoop   4096 May  4 04:54 docs
drwxr-xr-x.  7 hadoop hadoop     75 May  4 04:52 hbase-webapps
drwxr-xr-x.  3 hadoop hadoop   8192 May  4 04:52 lib
-rw-r--r--.  1 hadoop hadoop  11358 Jan 25 04:47 LICENSE.txt
-rw-r--r--.  1 hadoop hadoop    897 Feb 14 22:18 NOTICE.txt
-rw-r--r--.  1 hadoop hadoop   1477 Feb 12 19:21 README.txt
```

图 7.2　HBase 的文件目录

从图 7.2 中可以看到，根目录包含一些文本文件，如 CHANGES.txt(版本变更日志)、LICENSE.txt(许可条款)、NOTICE.txt(注意事项)和 README.txt(说明文档)。根目录中还包括如 bin、conf、docs、hbase-webapps 和 lib 目录，这些目录包含的内容如下。

1. bin

bin 目录中包含 HBase 提供的所有脚本，如 start-hbase. sh（启动 HBase 集群）、hbase-daemon. sh（启动或停止单个 HMaster/HRegionServer/ZooKeeper）、stop-hbase. sh（停止 HBase 集群）、hbase-config. sh（装载相关配置，如 HBASE_HOME 目录、HRegionServer 机器列表、JAVA_HOME 目录等）、hbase-env. sh（配置 JVM、GC 参数、LOG 目录）等。

2. conf

conf 目录中包含定义 HBase 的配置文件，如 hbase-site. xml（增加 HBase 的特定配置）、hbase-env. sh（配置 HBase 的环境变量）、regionservers（配置所有 HRegion 服务器的主机名）、log4j. properties（配置 HBase 的日志级别）等。

3. docs

docs 目录包含 HBase 工程网页的副本、工具、API 和项目自身的文档信息。读者可以用浏览器打开 docs/index. html 文件或 docs/book. html 文件查看有关 HBase 的更详细信息。

4. hbase-webapps

hbase-webapps 目录中包含 Java 实现的 Web 接口，用户在部署 HBase 到生产环境中或使用 HBase 时，一般很少会接触到该目录中的文件。

5. lib

lib 目录中包含 HBase 运行所依赖的很多类库，如 hadoop-client-2. 5. 1. jar、hadoop-hdfs-2. 5. 1. jar、hadoop-mapreduce-client-core-2. 5. 1. jar、hbase-server-1. 0. 0. jar 等类库。

7.4.3 配置 HBase

HBase 安装完成后，就要对其进行配置，根据 HBase 运行模式不同，其配置内容有所区别。HBase 的运行模式有单机模式（默认模式）、伪分布模式（所有守护进程都运行在单个节点上）和全分布模式（进程运行在物理服务器集群中）三种，读者可根据自己的集群规划来选择不同的运行模式，在这里采用全分布模式来对 HBase 进行配置。

1. 配置 hbase-env. sh

在配置文件 hbase-env. sh 里面可设置 HBase 环境变量，如可以配置 JVM 启动的堆大小、GC 的参数、HBase 的参数、Log 位置等。读者可以打开文件 conf/hbase-env. sh 细读其中的内容，每个选项都是有详细的注释的。本实例 hbase-env. sh 的配置内容如下。

```
#vim /opt/hbase-1.0.0/conf/hbase-env.sh
#配置内容：
export JAVA_HOME=/usr/java/jdk1.7.0_71/                #Java安装路径
export HBASE_LOG_DIR=${HBASE_HOME}/logs                #HBase日志路径
export HBASE_MANAGES_ZK=true                           #由 HBase负责启动和关闭 ZooKeeper
```

在一个分布式运行模式下的 HBase 需要依赖一个 ZooKeeper 集群，并且所有的节点和 Client 端都必须能够访问 ZooKeeper。默认情况下，HBase 会管理一个 ZooKeeper 集群，因此需要设置 HBASE_MANAGES_ZK 为 true，来由 HBase 管理启动和关闭。

2. 配置 hbase-site. xml

在配置文件 hbase-site. xml 中，可以通过设置 hbase. cluster. distributed 参数来设置 HBase 的运行模式。如果读者想了解有关 bhase-site. xml 的更多信息，请查看 HBase 项目源码目录 src/main/resources 下的 hbase-default. xml 文件。本实例 hbase-site. xml 的配置内容如下。

```
<configuration>
<property>
 <name>hbase.rootdir</name>
 <value>hdfs://master1.hadoop:9000/hbase</value>
</property>

<property>
 <name>hbase.cluster.distributed</name>
 <value>true</value>
</property>

<property>
 <name>hbase.zookeeper.quorum</name>
 <value>Slave1.Hadoop,Slave2.Hadoop,Slave3.Hadoop</value>
</property>

<property>
 <name>hbase.zookeeper.property.dataDir</name>
 <value>/opt/hbase-1.0.0/zookeeper</value>
</property>
</configuration>
```

其中，hbase. rootdir 是用来设置 HRegionServer 的共享目录，用来持久化 HBase，如果采用默认设置，HBase 会写到/tmp 目录下，当 HBase 重启时数据丢失；hase. cluster. distributed 用来设置 HBase 的运行模式，false 是单机模式，true 是分布式模式（包括伪分布式和全分布式），若为 false，HBase 和 ZooKeeper 会运行在同一个 JVM 中；hbase.

zookeeper.quorum 用来设置 ZooKeeper 集群的地址列表，地址之间用逗号分隔，如本实例将 Slave1.Hadoop、Slave2.Hadoop 和 Slave3.Hadoop 设置为 ZooKeeper 集群；hbase.zookeeper.property.dataDir 用来设置 ZooKeeper 保存数据的目录地址，这里设置为/opt/hbase-1.0.0/zookeeper 目录下。

3. 配置 regionservers

regionservers 文件用于配置运行 HBase 的机器，此文件的配置和 Hadoop 中的 Slave 配置类似，一行指定一台机器。本实例 regionservers 文件配置内容如下。

```
#vim /opt/hbase-1.0.0/conf/regionservers
#配置内容：
Slave1.Hadoop
Slave2.Hadoop
Slave3.Hadoop
```

4. 各主机间复制 HBase

本实例针对 HBase 的一系列配置都是在 Master1.Hadoop 的机器上进行配置的，经过上述对几个配置文件的配置，HBase 的配置已完成。最后，将 Master1.Hadoop 机器上的一系列配置文件复制到各个节点，从而完成 HBase 的分布式部署。本实例各主机间复制 HBase 的命令如下。

```
#scp -r /opt/hbase-1.0.0 Slave1.Hadoop:/opt
#scp -r /opt/hbase-1.0.0 Slave2.Hadoop:/opt
#scp -r /opt/hbase-1.0.0 Slave3.Hadoop:/opt
```

到此，HBase 的安装配置就已完成，在 HBase 的配置方面，只完成了最基本的配置，并没有对 HBase 进行优化配置，有兴趣的读者可以对 HBase 进行配置优化。

7.4.4 启停 HBase

HBase 安装配置完成之后，下一步来启动并测试 HBase。HBase 的正常启动顺序应该为 Hadoop→ZooKeeper→HBase，停止顺序为 HBase→ZooKeeper→Hadoop。因此，要启动 HBase，首先要保证 Hadoop 和 ZooKeeper 的正常运行。其中，只需要保证 Hadoop 的正常运行即可，ZooKeeper 的启停由 HBase 控制。HBase 的启动命令如下。

```
[hadoop@master1 /]$ start-hbase.sh
slave3.hadoop: starting zookeeper, logging to /opt/hbase-1.0.0/bin/../logs/hbase-hadoop-zookeeper
-slave3.hadoop.out
slave1.hadoop: starting zookeeper, logging to /opt/hbase-1.0.0/bin/../logs/hbase-hadoop-zookeeper
-slave1.hadoop.out
```




```
slave2.hadoop: starting zookeeper, logging to /opt/hbase-1.0.0/bin/../logs/hbase-hadoop-zookeeper
-Slave2.Hadoop.out
starting master, logging to /opt/hbase-1.0.0/logs/hbase-hadoop-master-master1.hadoop.out
slave2.hadoop: starting regionserver, logging to /opt/hbase-1.0.0/bin/../logs/hbase-hadoop-
regionserver-Slave2.Hadoop.out
slave1.hadoop: starting regionserver, logging to /opt/hbase-1.0.0/bin/../logs/hbase-hadoop-
regionserver-slave1.hadoop.out
slave3.hadoop: starting regionserver, logging to /opt/hbase-1.0.0/bin/../logs/hbase-hadoop-
regionserver-slave3.hadoop.out
```

HBase 启动成功后，可以通过 jps 命令来查看启动的 HBase 运行，在 Master 端 HBase 所启动的进程和 Slave 端 HBase 所启动的进程如图 7.3 所示。

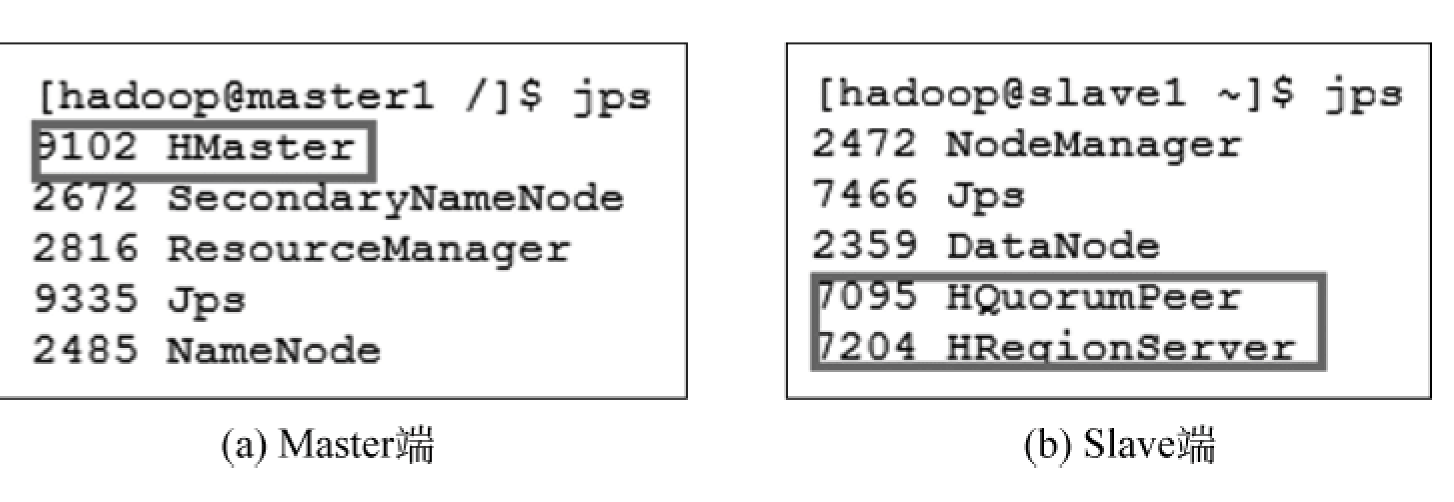


图 7.3　HBase 的启动进程

从图 7.3 中可以看出，在 Master 端和 Slave 端启动的 HBase 进程与表 7.3 所规划的 HBase 集群相同。其中，Master1.Hadoop 中启动的 HBase 进程为 HMaster；在 Slave1.Hadoop 中启动的 HBase 进程为 HQuorumPeer 和 HRegionServer。同理，在 Slave2.Hadoop 和 Slave3.Hadoop 中的 HBase 进程与 Slave1.Hadoop 中 HBase 进程相同。读者也可通过 HBase Master 基于 Web 的 UI 服务查看 HBase 集群的当前状态，如图 7.4 所示。

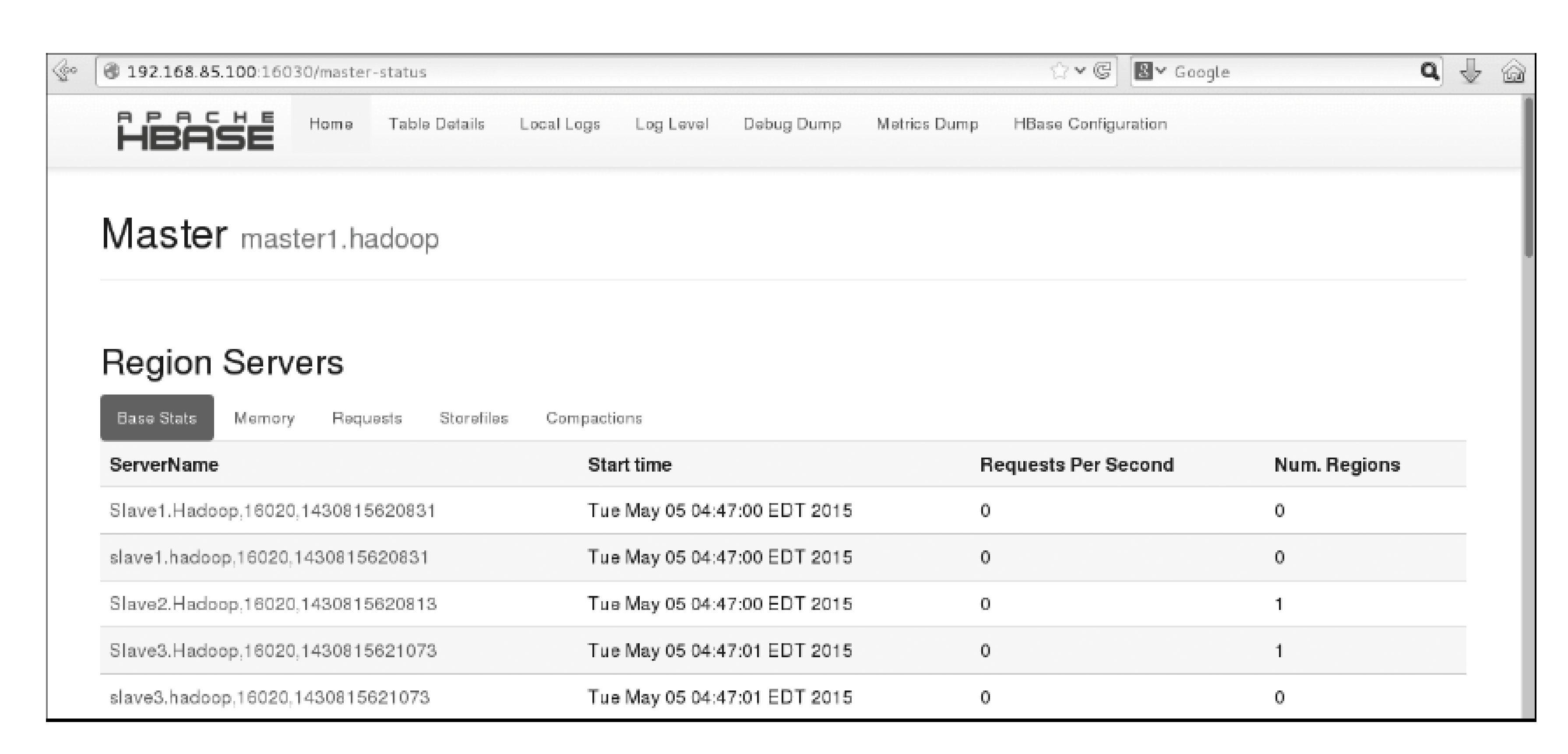


图 7.4　HBase Master 的 Web UI 界面

HBase Master 默认基于 Web 的 UI 服务端口为 16010，HBase HRegion 服务器默认基于 Web 的 UI 服务端口为 16030。本实例的 HMaster 运行在 Master1. Hadoop 的主机中，因此 HMaster 的主页地址就是 http://master1. hadoop:16010。如果要关闭 HBase 集群，可使用如下命令。

```
[hadoop@master1 ~ ]$ stop-hbase.sh
stopping hbase.....................
192.168.85.102: stopping zookeeper.
192.168.85.103: stopping zookeeper.
192.168.85.101: stopping zookeeper.
[hadoop@master1 ~ ]$ jps
4249 Jps
2557 NameNode
2932 ResourceManager
2775 SecondaryNameNode
[hadoop@master1 ~ ]$
```

本实例 HBase 集群中节点数据比较少，启动和关闭等待时间比较短，如果集群节点数量很多，则等待时间可能会很长。如果 HBase 是公布式运行模式，在关闭 Hadoop 集群之前一定要确认 HBase 是否已经被正常关闭了。

7.5 HBase 数据模型

HBase 不是一个关系型数据库，因此需要用不同的方法来定义数据模型。HBase 是以表的形式存储数据的，表由行和列组成，列又被划分为若干个列族/列簇（Column Family）组成，列簇支持动态扩展，不需要预定义数量及类型。

7.5.1 逻辑视图

HBase 实际上定义了一个四维数据模型（RowKey 行键、Column Family 列簇、Cell Qualifier 列修饰符和 Timestamp 时间戳）的表，并通过 HRegion 区域来存储数据。其中，HBase 中的表可以有上亿行、上百万列；该表是面向列的存储和权限控制；对于为空的列并不占用存储空间；每个 Cell Qualifier 中的数据可以有多个版本，默认情况下版本号是自动分配的，即为单元格插入时的时间戳；表中的数据都是字符串，没有类型。HBase 的逻辑视图如表 7.4 所示。

从表 7.4 可以看出，Key1、Key2 和 Key3 是三条记录的唯一 Row Key；Column Family1、Column Family2 是两个列簇；Column Family1 列簇下包括 Column1 和 Column2 两列，Column Family2 列簇下包括 Column1、Column2 和 Column3 三列；t1:hadoop 是由 Row Key1 和 Column1- Family1-Column1 唯一确定的一个单元格 Cell，该单元格中有一个数据为 hadoop，其中 t1 可看作自动分配的版本号。

表 7.4　HBase 逻辑视图

Row Key	Timestamp	Column Family1		Column Family2		
		Column1	Column2	Column1	Column2	Column3
Key1	t1	hadoop			kdf	
	t8	hdfs		ok		
	t2	map			sdf	
Key2	t4		hive			dfkd
	t7		data			
	t3		we		efde	
Key3	t5	df		sdf		
	t6	dfa			sdf	

1. Row Key 行键

在 HBase 表中，每行都有一个唯一的 Row Key，是表中每条记录的“主键”，该行键没有数据类型，可以使用任何字符串(最大长度是 64KB)作为行键。对于表中的行，可以根据行的键值进行排序，并且对表中行的访问都需要通过键值，可以通过单个 Row Key 访问、通过 Row Key 的排序访问或全表扫描的访问方式。

2. Column Family 列簇

HBase 表中的每列都归属于某个列簇，列簇必须作为表模式(Schema)定义的一部分预先给出，如 create 'ColumnFamily1'，'ColumnFamily2'。HBase 表中的列名以列簇作为前缀，每个列簇都可以有多个列成员，如 ColumnFamily1：Column1，ColumnFamily2：Column1；新加入的列簇成员可以随后按需、动态加入；同一列簇的数据存储在同一目录下，由几个文件保存。HBase 对表的权限控制、存储和调优都是在列簇层面进行的，即同一列簇成员有相同的访问模式和大小特征。

3. Cell Qualifier 列修饰符

列修饰符可以理解为实际的列名，可通过“列簇：列修饰符”描述具体的某个列，也可通过 Client 端随时把列添加到列簇中。

4. Cell 单元格

HBase 中通过 Row Key、Column Family 和 Column 唯一确定的一个存储单元为 Cell 单元格，其表示形式为：{Row Key，Column(=＜Family＞+＜Lable＞)，Timestamp}。Cell 单元格中的数据是没有类型的，全部是字节码形式存储的。

5. Timestamp 时间戳

HBase 表中的 Cell 单元格中保存着同一份数据的多个版本，版本号默认情况下是单元格插入时的时间戳(精确到毫秒的当前系统时间)。HBase 可根据唯一的时间戳来区分每个版本之间的差异。时间戳也可以由用户显式赋值，此时要注意生成的值要具有唯一性，从而避免数据版本冲突。

7.5.2 物理视图

表 7.4 为 HBase 中的一个虚表，仅是一个概念视图，并不是真实的存储形式，实际上表 7.4 在 HBase 中是按列簇进行存储的，每一个列簇为一个 Store，比如表 7.4 在 HBase 中的实际存储形式如图 7.5 所示。

Row Key	Timestamp	Clumn Family1
Key1	t1	Column1: hadoop
	t8	Column1: hdfs
	t2	Column1: map

(a)

Row Key	Timestamp	Clumn Family2
Key1	t1	Column1: kdf
	t8	Column1: ok
	t2	Column2: sdf

(b)

Row Key	Timestamp	Clumn Family1
Key2	t4	Column2: hive
	t7	Column2: data
	t3	Column2: we

(c)

Row Key	Timestamp	Clumn Family2
Key2	t4	Column3: dfld
	t3	Column2: efde

(d)

Row Key	Timestamp	Clumn Family1
Key3	t5	Column1: df
	t6	Column1: dfa

(e)

Row Key	Timestamp	Clumn Family2
Key3	t5	Column1: sdf
	t6	Column2: sdf

(f)

图 7.5 HBase 中表 7.4 的实际存储形式

从图 7.5 中可以看出，HBase 是面向列(Column-Oriented)的存储机制，每一个列簇都为单独的一张表，表 7.4 有三个 Row Key，即 Key1、Key2 和 Key3，并有两个列簇，在 HBase 中存储时就存为 6 张表，这里需要注意的是对于内容为空的列不进行存储。图 7.5(a)为 Row Key 为 Key1，列簇为 Column Family1 的表，Timestamp 值为数据的版本；图 7.5(b)为 Row Key 为 Key1，列簇为 Column Family2 的表；图 7.5(c)为 Row Key 为 Key2，列簇为 Column Family1 的表；图 7.5(d)为 Row Key 为 Key2，列簇为 Column Family2 的表；图 7.5(e)为 Row Key 为 Key3，列簇为 Column Family1 的表；图 7.5(f)为 Row Key 为 Key3，列簇为 Column Family2 的表。

7.6　HBase 关键技术

HBase 是一个分布式的、多版本的、面向列的开源数据库，利用 Hadoop HDFS 作为其文件存储系统，通过一系列的关键技术提供了高可靠性、高性能、列存储、可伸缩、实时读写的数据库系统。

7.6.1　HRegion 定位

当 Client 端在进行数据读写操作请求时，Client 本身并不知道哪个 HRegionServer 管理哪个 HRegion，都需要经过 HRegion 的定位来确定 Client 端所需要的数据到底在哪个 HRegionServer 上，并由 HRegionServer 实现数据的读写操作。HRegion 的定位过程主要涉及 HBase 内置的两张表"-ROOT-"表和".META"表。其中，".META"表包含所有的用户空间区域列表及 HRegionServer 的服务器地址，.META 可以有多个 HRegion；"-ROOT-"表包含.META 表所在的区域列表，用来查询所有.META 表中 HRegion 的位置，-ROOT-只有一个 HRegion。从存储结构和操作方法角度来说，这两张表与其他 HBase 的表没有任何区别，对普通表的操作都可应用于这两张表，并且这两张表的表结构相同，如表 7.5 所示。

表 7.5　"-ROOT-"和".META"表结构

RowKey	Info			Historian
TableName StartKey TimeStamp	RegionInfo	Server	Serverstartcode	
	StartKey,EndKey Family List	Address		

从表 7.5 中可以看出，RowKey 为 HRegion 的 Name，由 TableName、StartKey 和 TimeStamp 三部分组成，这三部分用逗号连接就构成了整个 RowKey，如一个 RowKey 的示例如下。

```
Table1,RK10000,12123132
```

表 7.5 中 info 包含 RegionInfo、Server 和 Serverstartcode。其中，RegionInfo 里包括 StartKey、EndKey 及每个 Family 的信息等；Server 存储的是管理这个 HRegion 的 HRegionServer 地址。HRegion 的定位使用三层类似于 B+树的结构来定位 HRegion 位置：第一层是 ZooKeeper 中包含的 RootRegion（只有一个，不进行拆分）位置信息的节点；第二层是从-ROOT-表中查找对应 Meta Region 的位置；第三层是从.META 表中查找用户表对应的 HRegion 的位置，HRegion 的具体定位过程如图 7.6 所示。

（1）Client 端首先连接到 ZooKeeper 管理的集群，并查询 ZooKeeper 中-ROOT-表，ZooKeeper 告诉 Client 端-ROOT-表中 HRegion 在哪个 HRegionServer 上，这里假定-ROOT-表中的 HRegion 在 RegionServer R 上。

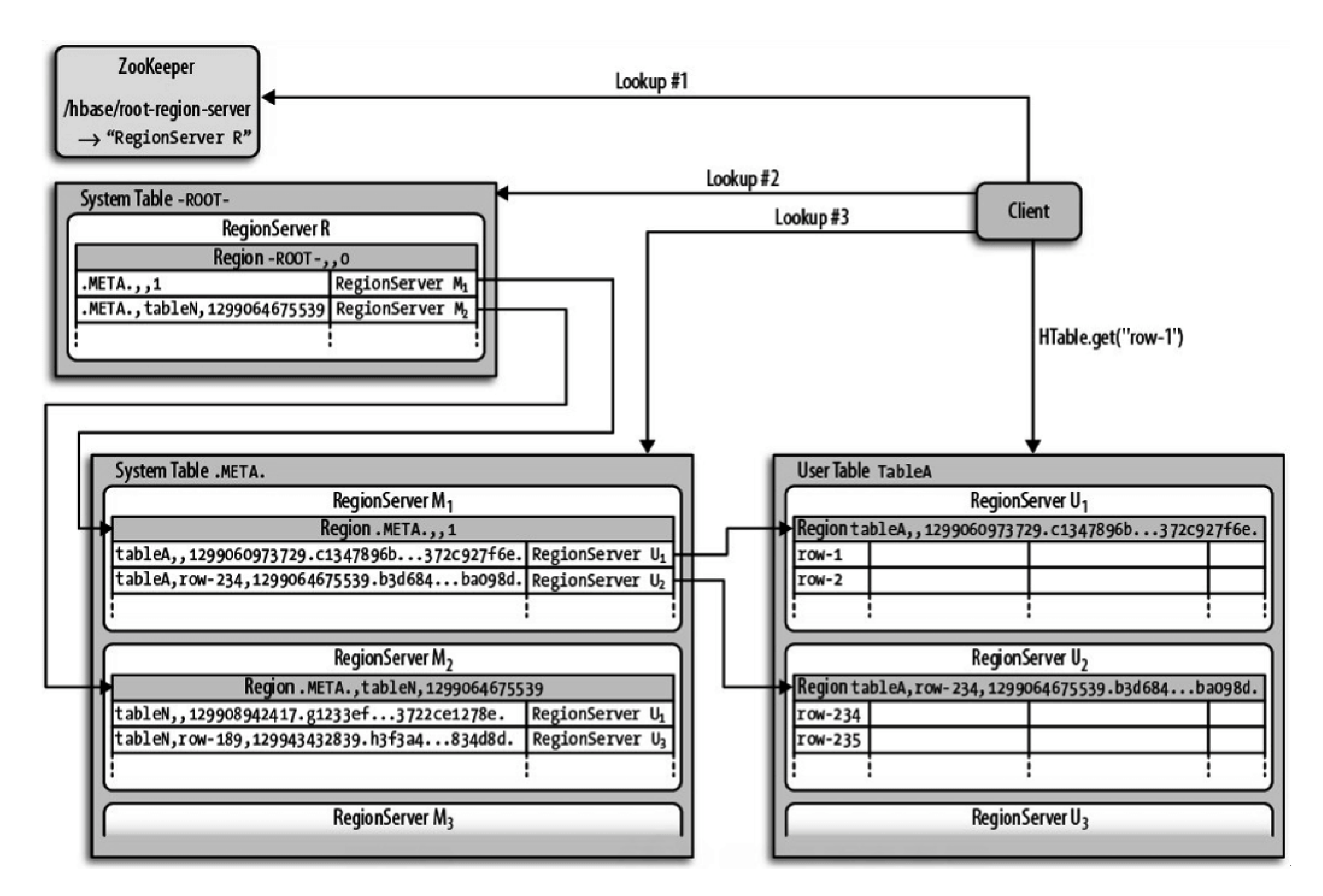


图 7.6 HRegion 定位过程

（图片来源：来自参考文献[81]）

（2）Client 获得 HRegionServer 的 Name 之后，再向 RegionServer R 发起查询某行的.META 的 HA 的 HRegion 可在某 HRegionServer 上找到，并将查询结果缓存到 Client 端。

（3）Client 端再向某 HRegionServer 上的.META 查询某行的数据在哪个 HRegion 上，并查询哪个 HRegionServer 可以提供服务，并将查询结果缓存到 Client 端，从而完成了 HRegion 定位。

如果读者想详细了解 HRegion 的定位过程，可以阅读 HBase 项目中 HConnectionManager 这个类，类中给出了 locateRegion()方法，该方法实现了 HRegion 的定位，其关键代码如下。

```
private HRegionLocation locateRegion(final TableName tableName,
        final byte [] row, boolean useCache, boolean retry){
        if(tableName.equals(TableName.META_TABLE_NAME)){
          return this.registry.getMetaRegionLocation();
        } else {
          //Region not in the cache-have to go to the meta RS
          return locateRegionInMeta (TableName.META_TABLE_NAME, tableName, row, useCache,
userRegionLock, retry);
```

```
        }
    }
```

其中，TableName.META_TABLE_NAME 就是-ROOT-表，在 HBase 0.96 版本里面它已经被取消了，取而代之的是 META 表中的第一个 regionHRegionInfo.FIRST_META_REGIONINFO，它位置在 ZooKeeper 的 meta-region-server 节点当中。如果需要查询的信息在缓存中，可直接访问相应的 HRegion，否则就要经过(1)～(3)过程，即 locateRegionInMeta()方法的实现过程。

注：从 HBase 0.96 版本开始，-ROOT-表已经改名为 hbase:namespace，.META 则是 hbase:meta，如果读者想了解更多的细节，请阅读 HBASE-8015 文档(https://issues.apache.org/jira/browse/HBASE-8015)。

7.6.2　HRegion 分裂

HRegion 是 HBase 实际存储数据的区域，其默认的 HRegion 的大小为 64MB。HBase 中每个表一开始只有一个 HRegion，随着数据不断地插入表中，HRegion 不断增大，当增大到某个阈值时，HRegion 会迅速自动分裂成两个 HRegion，并且这两个 HRegion 先会保存对原 HRegion 的引用，当这两个新 HRegion 数据拆分结束后，再将此引用去掉，并删除原有 HRegion。当表中行不断增多时，HRegion 数据也会逐渐增多。HRegion 自动分裂过程[82]如图 7.7 所示。

图 7.7 给出了 HRegion 自动分裂过程，其中红色线代表 HRegionServer 和 HMaster 的行为，绿色线代表 Client 行为。HRegion 具体分裂过程如下。

(1) HRegionServer 决定本地的 HRegion 是否需要分裂，如果需要分裂，则准备分裂工作：在 ZooKeeper 的/hbase/region-in-reansition/region-name 下创建一个 ZNode，并设为 SPLITTING 状态。

(2) HMaster 通过父 region-in-transition znode 的 watcher 监测到刚刚创建的 ZNode。

(3) HRegionServer 在 HDFS 中 RootHRegion 的目录下创建名为“.split”的子目录。

(4) HRegionServer 关闭 RootHRegion，并强制刷新缓存内的数据，并在本地数据结构中将标识改为下线状态，如果此时 Client 正好发送对 RootHRegion 的请求，将会抛出 NotServingRegionException 异常，Client 将重新尝试向其他的 HRegion 发送请求。

(5) HRegionServer 在.split 目录下为子 HRegionA 和 HRegionB 创建目录和相关的数据结构，然后 HRegionServer 分割 Store 文件(为 RootHRegion 的每个 Store 文件创建两个 Reference 文件，这些 Reference 文件将指向 RootHRegion 中的文件)。

(6) HRegionServer 在 HDFS 中创建实际的 HRegion 目录，并移动每个子 HRegion 的 Reference 文件。

(7) HRegionServer 向.META 表发送 Put 请求，并在.META 中将 HRegion 改为下线状态，添加子 HRegion 的信息，此时表中单独存储没有子 HRegion 信息的条目。Client 扫描.META 表时查看 RootHRegion 为分裂状态，但直到子 HRegion 信息出现在

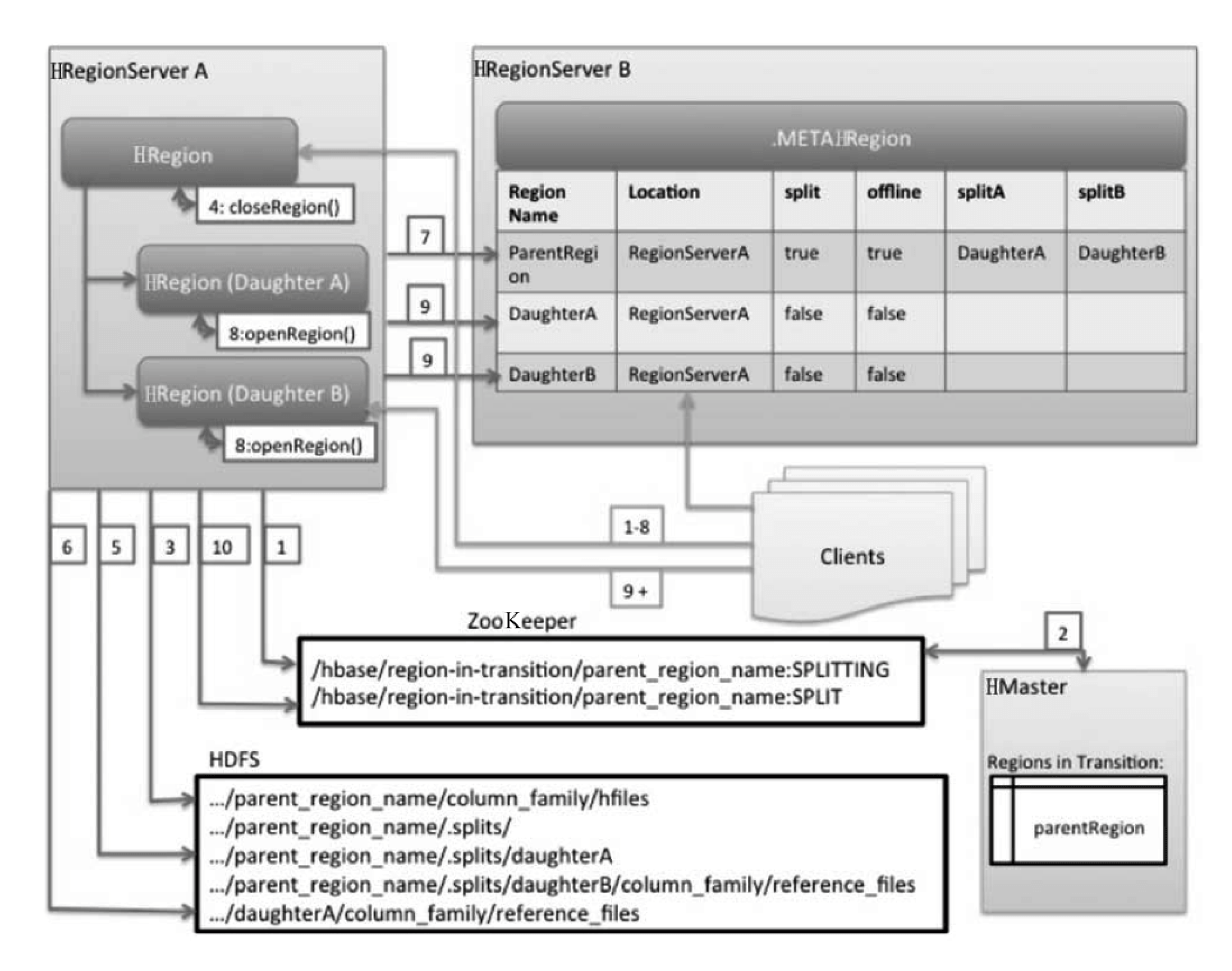


图 7.7 HRegion 自动分裂过程

（图片来源：来自参考文献[82]）

表中，Client 才知道它们的存在。如果 Put 请求成功，那么 RootHRegion 将被有效地分割；如果在这条 RPC 成功之前出现 HRegionServer 异常，那么 HMaster 和打开 HRegion 的下一个 HRegionServer 会清理关于该 HRegion 分裂的状态，并在.META 表更新之后，HRegion 的分裂将被 HMaster 回滚到之前的状态。

（8）HRegionServer 打开 HRegion，并行地接受写请求。

（9）HRegionServer 将子 HRegionA 和 HRegionB 的相关信息写入.META 表之后，Client 便可以扫描到新的 HRegion，并且可以向其发送请求。

（10）HRegionServer 将 ZooKeeper 中的 ZNode /hbase/region-in-transition/region-name 更改为 SPLIT 状态，以便 HMaster 可以监测到，如果子 HRegion 被选中，Balancer 可以自由地将子 HRegion 分派到其他 HRegionServer 上。

最后，HRegion 分裂之后，元数据和 HDFS 中依然包含着指向 RootHRegion 的 Reference 文件，这些 Reference 文件将在子 HRegion 发生紧缩操作重写数据文件时被删除掉，HMaster 的垃圾回收工会周期性地检测是否还有指向 RootHRegion 的 Reference，如果没有，将删除 RootHRegion。

在 HBase 0.94 版本之前，HRegion 的分裂策略采用 ConstantSizeRegionSplitPolicy，

如果要关闭 HRegion 的自动分裂，只需要将配置文件中的 hbase. hregion. max. filesize 设置为一个超大值。在 HBase 0. 94 版本之后，HRegion 的默认分裂策略是 IncreasingToUpperBoundRegionSplitPolicy，这个策略的核心思想是当 HRegion 的大小超过某个阈值时，HRegion 就进行自动分裂。这个阈值主要由 hbase. hregion. max. filesize、hbaase. increaing. policy. initial. size 和当前在一个 HRegionServer 中同一个表的 HRegion 数量个数有关。如果读者希望实现自己的 HRegion 分裂策略，首先需要继承 RegionSplitPolicy，再实现 shouldSplit ( )、getSplitPoint ( ) 方法和 HRegionServer 的 splitRegion()方法等。

7.6.3　HBase 读写机制

HBase 读写数据过程都是由 HRegionServer 负责的，HRegionServer 一方面与 Client 进行交互，负责处理用户的读写请求，提供对 HRegion 的管理和服务；另一方面与 HMaster 进行交互，上传 HRegion 的负载信息，参与 HMaster 的分布式协调管理。因此，HBase 的数据读写过程涉及 HRegionServer、HMaster、HRegion 和 Client。其中，HRegionServer 是 HBase 中读写数据的核心，主要负责响应用户的 I/O 请求，并向 HDFS 文件系统中读写数据。HRegionServer 与 HMaster 之间通过 RPC 通信协议进行通信，在此过程中 HMaster 扮演了 RPC Server 的角色，HRegionServer 扮演了 RPC Client 的角色，主要负责定期向 HMaster 汇报自身的内存使用状态、HRegion 在线状态等信息。HRegionServer 与 Client/HRegion 之间也是采用 RPC 通信协议进行通信，在此过程中 Client/HRegion 扮演 RPC Client 角色，而 HRegionServer 扮演 RPC Server 角色，主要负责数据更新、读取、删除、写入等操作。HBase 的读写过程如图 7.8 所示。

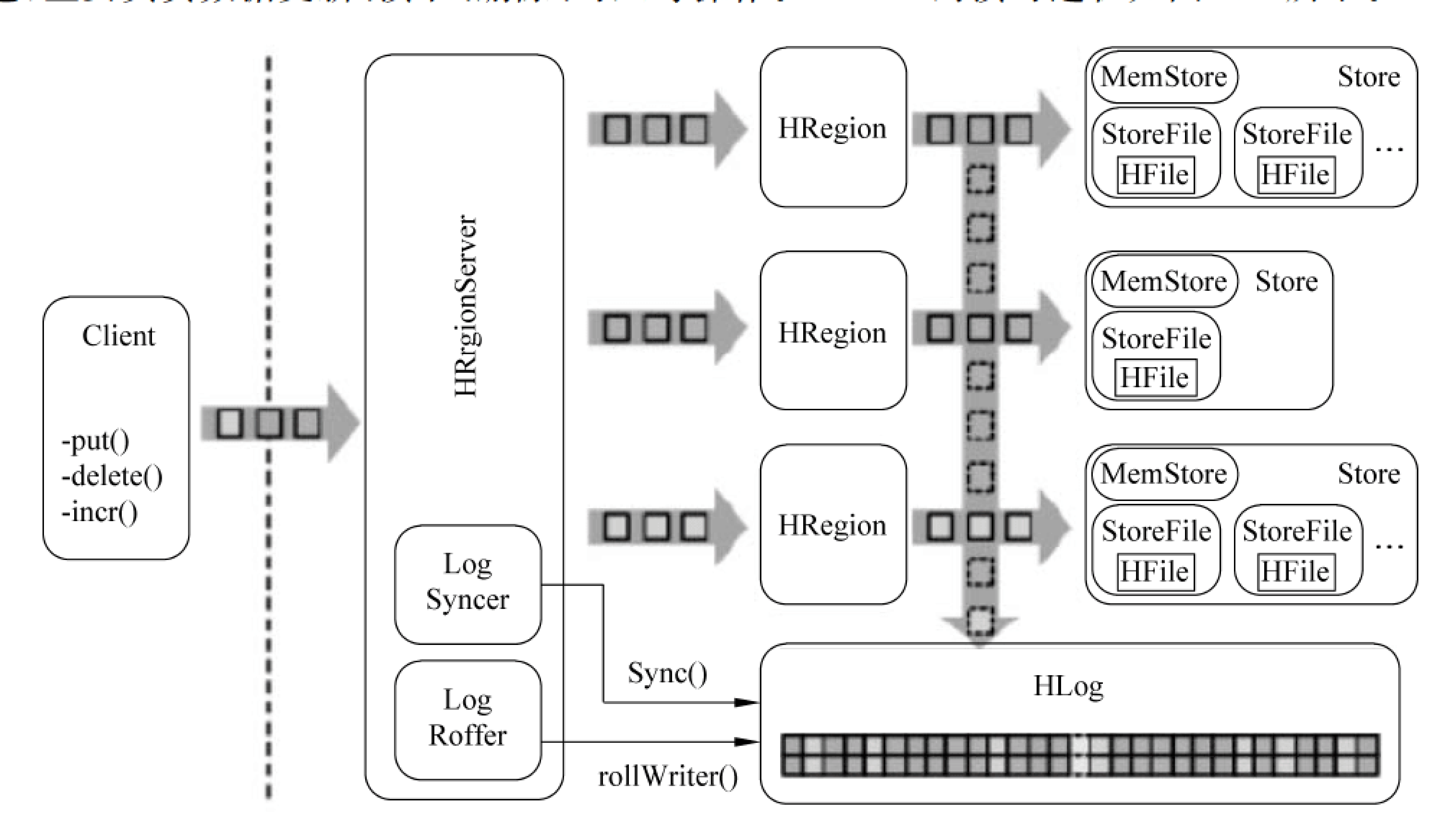


图 7.8　HBase 的读写过程

用户使用 HBase 写入或更新数据的整个过程如下。

（1）Client 向 HRegionServer 启动一个操作来进行写入或更新数据操作，如 put()请求。

（2）Client 端每次对数据的修改都封装到一个 KeyValue 对象实例中，通过 RPC 调用(含有匹配的 HRegion 的 HRegionServer)发送给 HRegionServer。

（3）HRegionServer 收到 KeyValue 实例后，KeyValue 会被分配到对应的 HRegion 实例，并将数据写入到 HLog 中（WAL 存储格式），然后再放入实际拥有记录的 MemStore 中。

（4）当 MemStore 达到一定大小或经历某个特定时间时，数据会异步连续写入到 HFile 中。

为了防止在数据写入过程中，存储在内存中的数据没来得及保存到磁盘而造成的数据丢失情况，HBase 使用 WAL 格式的日志 HLog 存储在 HDFS 上。当 HRegionServer 出现异常时，其他 HRegionServer 也可以读取日志文件并回放修改，从而恢复数据，并能保证数据不丢失。

当用户使用 HBase 读取数据时，如果所要读取的数据所在的 HRegionServer 地址在 Cache 缓存中时，Client 端可以直接定位到该 HRegionServer，然后直接在该 HRegionServer 的某个 HRegion 上查找到匹配的数据。否则 HBase 的读数据流程需要经过以下过程。

（1）Client 从 ZooKeeper 获取.META 所在的 HRegionServer 地址；

（2）根据 RowKey 查询所在的 HRegion，获得 HRegion 所在的 HRegionServer 地址；

（3）Client 连接该 HRegionServer，从而查找到匹配的数据。

HBase 的顺序读取速度非常快，这是由架构本身和底层的数据结构共同决定的，即由 LSM-Tree(Log-Structured Merge-Tree)、HRegion 分区和 Cache 缓存共同决定。其中，LSM 树状结构相比 B/B+树而言，磁盘的顺序读取速度很快；HBase 读取数据首先会在 Cache 缓存中采用 LRU(最近最少算法)查找，如果缓存中没有找到，会从内存中的 MemStore 中查找，只有这两个地方都没有找到时，才会加载 HFile，某种程度上也减少和节省了寻道时间的开销。因此，HBase 的顺序读取速度非常快。

7.7 HBase 交互接口

HBase 本身是由 Java 编写的，因此 HBase 服务器端需要 JVM 的支持，但是对于客户端来说，不仅可以通过使用 Java 客户端 API 和 HBase 附带的函数库与 HBase 进行交互，HBase 还提供了其他不使用 Java 的客户端与 HBase 进行交互，如 HBase Shell、HBase Thrift、REST Gateway、MapReduce、Hive 和 Pig 等方式与 HBase 交互。其中，Native Java API 是一种最常规和高效的访问方式，适合于 Hadoop MapReduce Job 并行批处理的 HBase 表数据；HBase Shell 为 HBase 的命令行工具，最简单的接口，适合 HBase 管理使用；HBase Thrift 是利用 Thrift 序列化技术，支持 C++、PHP、Python 等多种语言，适合其他异构系统在线访问 HBase 表数据；REST Gateway 支持 REST 风格的 HTTP API 访问 HBase，解除了语言限制；MapReduce 可直接使用 MapReduce 作业处理

与 HBase 交互;Hive 相当于传统的数据仓库,使用类似 SQL 语言来访问 HBase;Pig 是一种可使用 Pig Latin 流式编程语言来操作 HBase 中的数据,适合做数据统计。下面介绍两种比较常用的与 HBase 交互的方式。

7.7.1　Native Java API

Native Java API 是比较常用的与 HBase 进行交互的一种方式,这种方式需要在集群中创建 Java 项目,并调用 HBase 的 API 来操作 HBase,主要涉及对 HBase 的创建表格、删除表格、插入数据、删除数据、查询数据等操作。Native Java API 操作的具体流程包括创建项目、获取 jar 包到项目的 lib 目录下(一般将 HBase 的 lib 目录下的所有 jar 包添加到项目工程目录 lib 下)和编写 Java 程序等过程。下面重点介绍如何通过编写 Java 应用程序与 HBase 进行交互。要编写 Java 应用程序与 HBase 进行交互,首先要了解 HBase 的数据模型与 Native Java API 类之间的对应关系(如表 7.6 所示)。

表 7.6　HBase 数据模型与 Java 类之间的对应关系

Java 类	HBase 数据模型
HBaseAdmin	DataBase(数据库)
HBaseConfiguration	
HTable	Table(表)
HTableDescriptor	Column Family(列簇)
Put	Column Qualifier(列修饰符)
Get	
Scan	

注:有关 HBaseAdmin、HBaseConfiguration、HTable、HTableDescriptor、Put、Get、Scan 等类的详细说明和用法,请查看 HBase API(http://hbase.apache.org/apidocs/overview-summary.html)。

1. HBaseConfiguration

HBaseConfiguration 主要用于对 HBase 进行配置。HBaseConfiguration 提供的具体方法及说明如表 7.7 所示。

表 7.7　HBaseConfiguration 提供的方法

返回值	方　　法	描　　述
void	addResource(Path file)	通过给定路径所指的文件添加资源
	clear()	清空所有已设置的属性
	set(String name, String value)	通过属性名来设置值
	setBoolean(String name, boolean value)	设置 boolean 类型的属性值

续表

返回值	方　法	描　述
String	getBoolean(String name, boolean defaultValue)	获取为 boolean 类型的属性值，如果其属性值类型不为 boolean，则返回默认属性值
	get(String name)	获取属性名对应的值

HBaseConfiguration 的用法如下所示。

```
HBaseConfiguration config=new HBaseConfiguration();
//可以自定义配置,也可以从自定义配置文件中读取
/* config.set("hbase.zookeeper.property.clientPort", "2181");
config.set("hbase.zookeeper.quorum", "Slave1.Hadoop,Slave2.Hadoop,Slave3.Hadoop");
config.set("hbase.master", "Master1.Hadoop \\:600000"); * /
```

2. HBaseAdmin

HBaseAdmin 提供了一个接口来管理 HBase 数据库存的表信息，如创建表、删除表、列出表项、使表有效或无效、添加或删除表列簇成员等。HBaseAdmin 提供的具体方法如表 7.8 所示。

表 7.8　HBaseAdmin 提供的方法

返 回 值	方　法	描　述
void	addColumn(String tableName, HColumnDescriptor column)	对已经存在的表添加列
	checkHBaseAvailable (HBaseConfiguration conf)	查看 HBase 是否处于运行状态
	createTable (HTableDescriptor desc)	创建一个表，同步操作
	deleteTable(byte[] tableName)	删除一个已经存在的表
	enableTable(byte[] tableName)	使表处于有效状态
	disableTable(byte[] tableName)	使表处于无效状态
	modifyTable(byte[] tableName, HTableDescriptor htd)	修改表模式，异步操作，可能需要花费一定的时间
HTableDescriptor	listTables()	列出所有用户控件表项
boolean	tableExists(String tableName)	检查表是否存在

HBaseAdmin 的用法如下所示。

```
HBaseAdmin admin=new HBaseAdmin(config);
//创建表
HTableDescriptor htd=new HTableDescriptor(tableName);
```

```
htd.addFamily(new HColumnDescriptor("cf1"));
htd.addFamily(new HColumnDescriptor("cf2"));
admin.createTable(htd);
//修改表信息
admin.disableTable(tableName);
//modifying existing ColumnFamily
admin.modifyColumn(tableName, new HColumnDescriptor("name"));
admin.enableTable(tableName);
//删除表
admin.disableTable(Bytes.toBytes(tableName));
admin.deleteTable(Bytes.toBytes(tableName));
```

3. HTableDescriptor

HTableDescriptor 用于维护列簇信息，如版本号、压缩设置等，通常在创建表或者为表添加列簇时使用。有关 HTableDescriptor 提供的具体方法及说明如表 7.9 所示。

表 7.9　HTableDescriptor 提供的方法

返回值	方　　法	描　　述
byte[]	getName()	获取列簇的名字
	getValue(byte[] key)	获取对应的属性的值
void	setValue(String key, String value)	设置对应属性的值

HTableDescriptor 的用法如下所示。

```
//添加一个 content 列簇
HTableDescriptor htd=new HTableDescriptor(tablename);
HColumnDescriptor col=new HColumnDescriptor("content:");
htd.addFamily(col);
```

4. HTable

HTable 主要用于和 HBase 直接交互，如获取值和表名、检查表是否有效、向表中添加值等操作。有关 HTable 所提供的具体方法及说明如表 7.10 所示。

表 7.10　HTable 提供的方法

返 回 值	方　　法	描　　述
void	checkAdnPut(byte[] row, byte[] family, byte[] qualifier, byte[] value, Put put)	自动检查 row/family/qualifier 是否与给定的值匹配
	close()	释放所有的资源或挂起内部缓冲区中的更新
	put(Put put)	向表中添加值

续表

返回值	方　法	描　述
Boolean	exists(Get get)	检查 Get 实例所指定的值是否存在于 HTable 的列中
Result	get(Get get)	获取指定行某些单元格所对应的值
byte[][]	getEndKeys()	获取当前已打开的表每个区域的结束键值
ResultScanner	getScanner(byte[] family)	获取当前给定列簇的 scanner 实例
HTableDescriptor	getTableDescriptor()	获取当前表的 HTableDescriptor 实例
byte[]	getTableName()	获取表名
static boolean	isTableEnabled (HBaseConfiguration conf，String tableName)	检查表是否有效
void	put(Put put)	向表中添加值

HTable 的用法如下所示。

```
HTable htable= ...                                   //instantiate HTable
Scan scan= new Scan();
scan.addColumn(Bytes.toBytes("cf"),Bytes.toBytes("attr"));
scan.setStartRow(Bytes.toBytes("row"));              //start key is inclusive
scan.setStopRow(Bytes.toBytes("row"+    (char)0));
                                                     //stop key is exclusive
ResultScanner rs= htable.getScanner(scan);
try {
  for(Result r= rs.next(); r != null; r= rs.next()){
  //process result...
} finally {
  rs.close();                                        //always close the ResultScanner!
}}
```

5. Put

Put 用来对单个行执行添加操作，有关 Put 所提供的具体方法及说明如表 7.11 所示。

表 7.11　Put 提供的方法

返回值	方　法	描　述
Put	add(byte[] family，byte[] qualifier，byte[] value)	将指定的列和对应的值添加到 Put 实例中
	add(byte[] family，byte[] qualifier，long ts，byte[] value)	将指定的列和对应的值及时间戳添加到 Put 实例中
	setTimeStamp(long timeStamp)	设置 Put 实例的时间戳

续表

返 回 值	方　　法	描　　述
byte[]	getRow()	获取 Put 实例的行
RowLock	getRowLock()	获取 Put 实例的行锁
long	getTimeStamp()	获取 Put 实例的时间戳
boolean	isEmpty()	检查 familyMap 是否为空

Put 的用法如下所示。

```
//向表 tablename 添加 family,qualifier,value 指定的值
HTable table=new HTable(conf,Bytes.toBytes(tablename));
Put put=new Put(rowkey);
put.add(family,qualifier,value);
table.put(put);
```

6. Get

Get 用来获取对单个行执行添加操作，有关 Get 提供的具体方法及说明如表 7.12 所示。

表 7.12　Get 提供的方法

返回值	方　　法	描　　述
Get	addColumn(byte[] family，byte[] qualifier)	获取指定列簇和列修饰符对应的列
	addFamily(byte[] family)	通过指定列簇获取其对应的所有列
	setTimeRange (long minStamp， long maxStamp)	获取指定列的版本号
	setFilter(Filter filter)	当执行 Get 操作时设置服务器端的过滤器

Get 的用法如下所示。

```
//获取 tablename 表中 row 行的对应数据
HTable table=new HTable(config,Bytes.toBytes(tablename));
Get get=new Get(Bytes.toBytes(row));
Result result=table.get(get);
```

7. Scan

Scan 用来查询表的相关信息及数据，有关 Scan 提供的具体方法及说明如表 7.13 所示。

表 7.13 Scan 提供的方法

返回值	方法	描述
Scan	addFamily()	从指定列簇中查询所有列
	addColumn()	查询指定列簇的某列
	setMaxVersions()	指定最大的版本个数
	setTimeRange()	指定最大的时间戳和最小的时间戳
	setTimeStamp()	指定时间戳
	setFilter()	指定 Filter 来过滤掉不需要的信息
	setStartRow()	指定开始的行
	setStopRow()	指定结束的行(不含此行)
	setBatch()	指定最多返回的 Cell 数目

Scan 的用法如下所示。

```
//获取 tablename 表中 row 行的对应数据
HTable table= (HTable)getHTablePool().getTable(tablename);
Scan scan= new Scan();
scan.setMaxVersions();
//指定最多返回 Cell 数目,防止一行中有过多的数据,导致 OutofMemory 错误
scan.setBatch(1000);
//scan.setTimeStamp(NumberUtils.toLong("1234567890"));
//scan.setTimeRange(NumberUtils.toLong("1234567890"),
NumberUtils.toLong("1370336337163"));
//scan.setStartRow(Bytes.toBytes("startRow"));
//scan.setStopRow(Bytes.toBytes("stopRow"));
//scan.addFamily(Bytes.toBytes("family"));
//scan.addColumn(Bytes.toBytes(family), Bytes.toBytes(column));
//查询 family 列簇,列 column 值为 1 的记录
//方法一 (单个查询)
//Filter filter= new SingleColumnValueFilter(
//Bytes.toBytes("family"),Bytes.toBytes("column"),CompareOp.EQUAL,Bytes.toBytes("1"));
//scan.setFilter(filter);
```

有关 Native Java API 与 HBase 的交互实现，请查看本书配套资料中 HadoopWorksapces 文件夹下的 HBase 项目，该项目首先通过 HBase API 创建一个名为 user 的表，该表中有两个列簇为 name 和 article，name 列簇有 name 和 nickname 两个列，article 列簇有 title、content 和 tag 三个列。通过 HBase API 对 user 表进行添加数据、根据 rowkey 查询、遍历查询 user 表、更新表中的某一列、删除指定的列、删除表等操作。请读者认真阅读源码，并理解和学会使用 HBase API 创建表、删除表、记录信息的增删查改等操作。

7.7.2　HBase Shell

HBase Shell 是基于 Ruby Shell 实现的，是 HBase 集群的命令行接口，用户可以使用 Shell 访问本地或远程服务器并与其进行交互，而且 Shell 还提供了 Client 端和管理功能的操作。HBase Shell 常用的操作命令如表 7.14 所示。

表 7.14　HBase Shell 常用命令

Shell 命令	名　称
create	创建表
describe	显示表相关的详细信息
count	统计表中行的数量
alter	修改列簇模式
delete	删除指定表/行/列对应的值
deleteall	删除指定行的所有元素
disable	使表无效
drop	删除表
enable	使表有效
exists	测试表是否存在
exit	退出 HBase Shell
get	获取行或单元(Cell)的值
list	列出 HBase 中存在的所有表
put	向指定的表单元添加值
tools	列出 HBase 所支持的工具
scan	通过对表的扫描来获取对应的值
status	返回 HBase 集群的状态信息
shutdown	关闭 HBase 集群
truncate	重新创建指定表
version	返回 HBase 版本信息

HBase Shell 的这些常用命令的具体操作，可通过 help 查看到详细的用法。首先在保证 Hadoop 正常运行的情况下启动 HBase，HBase 启动成功后，执行如下命令。

```
[hadoop@Master1 ~ ]$ hbase shell
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
Version 1.0.0, r6c98bff7b719efdb16f71606f3b7d8229445eb81, Sat Feb 14 19:49:22 PST 2015
hbase(main):001:0>
```

HBase Shell 启动成功之后，可以输入 help 命令回车，会返回文本的帮助信息，如下所示(本示例省略了部分内容)。

```
hbase(main):002:0>help
HBase Shell, version 1.0.0, r6c98bff7b719efdb16f71606f3b7d8229445eb81, Sat Feb 14 19:49:22 PST 2015
Type 'help "COMMAND"', (e.g. 'help "get"' -- the quotes are necessary) for help on a specific command.
    ...
The HBase shell is the (J)Ruby IRB with the above HBase-specific commands added.
For more on the HBase Shell, see http://hbase.apache.org/book.html
hbase(main):003:0>
```

这里为了给读者演示，假定需要创建一个名为 person 的表，该表有两个列簇，分别为 name 和 address，name 只有一个，address 可以有多个，具体的 Shell 命令如下。

```
hbase(main):003:0>create 'person','name','address'
0 row(s) in 3.0080 seconds
=>Hbase::Table-person
```

向 person 表中插入一条记录，记录的 row 值为 row1，列簇 name 的值为 tom，具体的 Shell 命令如下。

```
hbase(main):004:0>put 'person','row1','name','tom'
0 row(s) in 0.3700 seconds
```

向 person 表的 address 列簇分别插入家庭地址 home 列和工作地址 work 列，并查询其家庭地址，具体的 Shell 命令如下。

```
hbase(main):005:0>put 'person','row1','address:home','xian street'
0 row(s) in 0.0640 seconds

hbase(main):006:0>put 'person','row1','address:work','shanghai street'
0 row(s) in 0.0260 seconds

hbase(main):007:0>get 'person','row1',{COLUMN=>'address:home'}
COLUMN                    CELL
address:home             timestamp=1431503150371, value=xian street
1 row(s) in 0.2610 seconds
```

在 person 表中，根据 row 值更新 name 值，首先根据 row 值查询当前的 name 值，再执行 put 命令后，查看该 row 值更新后的 name 值，具体命令如下。

```
hbase(main):008:0>get 'person','row1'
COLUMN                    CELL
```




```
address:home          timestamp=1431503150371, value=xian street
address:work          timestamp=1431503185691, value=shanghai street
name:                  timestamp=1431502932220, value=tom
3 row(s)in 0.0420 seconds

hbase(main):009:0>put 'person','row1','name','jack'
0 row(s)in 0.0370 seconds

hbase(main):010:0>get 'person','row1',{COLUMN=>'name',TIMESTAMP=> 1431502932220}
COLUMN                                        CELL
name:                                  timestamp=1431502932220, value=tom
1 row(s)in 0.0350 seconds
```

通过 list 命令查询已创建的表信息；通过 describe 命令返回 person 表的详细信息，包括列簇的列表；查询 person 表中所有数据；查看 person 表中 address 列簇的所有数据等操作，具体的命令如下所示。

```
//通过 list 命令查询已创建的表信息
hbase(main):011:0>list
TABLE
person
1 row(s)in 0.0420 seconds
=>["person"]

//通过 describe 命令返回 person 表的详细信息
hbase(main):012:0>describe 'person'
Table person is ENABLED
person
COLUMN FAMILIES DESCRIPTION
{NAME=>'address', DATA_BLOCK_ENCODING=>'NONE', BLOOMFILTER=>'ROW', REPLICATION_SCOPE=>'0',
VERSIONS=>'1', COMPRESSION=>'NONE', MIN_VERSIONS=>
'0', TTL=>'FOREVER', KEEP_DELETED_CELLS=>'FALSE',
BLOCKSIZE=>'65536', IN_MEMORY=>'false', BLOCKCACHE=>'true'}
{NAME=>'name', DATA_BLOCK_ENCODING=>'NONE', BLOOMFILTER=>'ROW', REPLICATION_SCOPE=>'0',
VERSIONS=>'1', COMPRESSION=>'NONE', MIN_VERSIONS=>'0', TTL=>
'FOREVER', KEEP_DELETED_CELLS=>'FALSE', BLO
CKSIZE=>'65536', IN_MEMORY=>'false', BLOCKCACHE=>'true'}
2 row(s)in 0.2190 seconds

//查询 person 表中所有数据
hbase(main):013:0>scan 'person'
ROW                COLUMN+CELL
```

```
 row1     column=address:home, timestamp=1431503150371, value=xian street
 row1     column=address:work, timestamp=1431503185691, value=shanghai street
 row1     column=name:, timestamp=1431504003274, value=jack
1 row(s)in 0.0350 seconds

//查看 person 表中 address 列簇的所有数据
hbase(main):014:0>scan 'person',{COLUMNS=>'address'}
ROW
 row1      column=address:home, timestamp=1431503150371, value=xian street
 row1      column=address:work, timestamp=1431503185691, value=shanghai street
1 row(s)in 0.0450 seconds

//查询 person 表是否存在
hbase(main):015:0>exists 'person'
Table person does exist
0 row(s)in 0.0330 seconds

//查询服务器状态
hbase(main):016:0>status
3 servers, 0 dead, 1.0000 average load

//查询 HBase 版本
hbase(main):017:0>version
1.0.0, r6c98bff7b719efdb16f71606f3b7d8229445eb81, Sat Feb 14 19:49:22 PST 2015

//查询 person 表是否 enable
hbase(main):018:0>is_enabled 'person'
true
0 row(s)in 0.0710 seconds

//查询 person 表是否 disable
hbase(main):019:0>is_disabled 'person'
false
0 row(s)in 0.0730 seconds
```

对 person 表删除整行，清空整张表，设置表不可用，删除表，关闭 Shell 等操作的具体命令如下所示。

```
//删除 person 表中的一整行
hbase(main):020:0>deleteall 'person','jack'
0 row(s)in 0.0230 seconds
//清空 person 表
hbase(main):021:0>truncate 'person'
```




```
Truncating 'person' table(it may take a while):
 - Disabling table...
 - Truncating table...
0 row(s) in 2.2380 seconds

//设置 person 表不可用
hbase(main):022:0> disable 'person'
0 row(s) in 1.4280 seconds

//删除 person 表
hbase(main):023:0> drop 'person'
0 row(s) in 0.3700 seconds

hbase(main):024:0> list
TABLE
0 row(s) in 0.0230 seconds
=> []

//关闭 Shell
hbase(main):025:0> exit
[hadoop@Master1 ~]$
```

上面通过 HBase Shell 为读者演示了创建、查询、更新、删除等操作，有关更多的 Shell 命令请读者通过 help 命令或查看相关的文档了解。

7.8　HBase 快照机制

HBase 快照（HBase Snapshots）是对一个表进行快照，快照不是表的复制，而是一个文件名称列表，因而不会复制数据。快照恢复是指恢复到之前的“表结构”以及当时的数据，快照之后发生的数据不会恢复。HBase 0.95 版本之前，HBase 快照功能默认是关闭的；从 HBase 0.95 版本以后，该功能默认是开启的。当需要从应用异常中还原或从一个已知的安全状态恢复等情况下，都可以使用 HBase 快照。HBase 快照的常用操作如下。

1. 生成快照

本操作可尝试对指定表生成快照，如果集群在执行数据均衡、分隔或合并等操作时，可能会对生成快照操作造成影响。当要执行生成快照命令时，首先需要检查 HBase 的快照功能是否已开启。如果要开始该功能，需要修改 hbase-site.xml 配置文件中的配置参数，修改内容如下所示。

```
<property>
   <name>hbase.snapshot.enabled</name>
   <value>true</value>
</property>
```

如要对 person 表进行快照操作,需要进入 HBase Shell 使用快照命令 snapshot,具体的操作内容如下。

```
hbase(main):005:0> snapshot 'person','snapshot_person_20150514'
0 row(s) in 1.2420 seconds
```

快照生成之后,可以使用 list_snapshots 命令列出所有的快照,并且会显示出快照名称、源表以及创建日期和时间信息,具体命令如下:

```
hbase(main):006:0> list_snapshots
SNAPSHOT                          TABLE+ CREATION TIME
snapshot_person_20150514          person(Thu May 14 13:44:51+0800 2015)
1 row(s) in 0.0490 seconds

=> ["snapshot_person_20150514"]
```

2. 克隆快照

本操作使用与指定快照相同的结构数据构建一张新表,会生成一张有完整功能的表,对该表的任意修改都不会对原表或快照产生影响。克隆快照使用 clone_snapshot 命令从指定的快照生成新表,该操作不会产生数据复制。克隆快照的具体命令如下。

```
hbase(main):007:0> clone_snapshot 'snapshot_person_20150514','colensnapshot_person'
0 row(s) in 0.8520 seconds
```

3. 还原快照

如果需要将快照替换为当前表结构/数据,可使用 restore_snapshot 命令来还原快照,具体命令如下。

```
hbase(main):008:0> restore_snapshot 'snapshot_person_20150514'
0 row(s) in 0.8310 seconds
```

4. 导出快照

如果需要将快照导出至其他集群,可使用 ExportSnapshot 工具将现有快照导出至其他集群。导出工具只是将快照数据和元数据复制到其他集群,操作只会涉及 HDFS,不会影响到集群的负载。导出快照的具体命令如下。

```
[hadoop@Master1 ~]$ hbase org.apache.hadoop.hbase.snapshot.ExportSnapshot
-snapshot 'snapshot_person_20150514' -copy-to hdfs://master1.hadoop:8082/hbase
```

5. 删除快照

当不再需要指定表的快照时，可以对该快照进行删除操作，并且释放未共享的磁盘空间，该操作不会影响其他克隆或快照。删除快照可使用 delete_snapshot 命令，如下所示。

```
hbase(main):009:0> delete_snapshot 'snapshot_person_20150514'
0 row(s) in 0.0220 seconds
```

当前 HBase 快照包含所有的基础功能特性，但是还不完善，如当执行复制快照命令时还原一个表，可能会造成两个集群不同步等异常情况的发生。相信在往后的 HBase 版本中，快照的功能会有所完善。



第8章 数据仓库 Hive

Hive 是基于 Hadoop 构建的一套数据仓库分析系统，它提供了丰富的类 SQL 查询方式来分析存储在 Hadoop 分布式文件系统中的数据。本章将重点介绍 Hive 的数据模型、数据类型、Hive 基本操作以及内置运算符和函数等内容。

8.1 Hive 概述

Hive 是一个基于 Hadoop 的数据仓库基础架构，它提供了数据存储管理和大型数据集的查询与分析的一系列工具，能够很方便地对存储于 Hadoop 文件系统中的数据进行简单的数据汇总、查询、分析等。Hive 提供了一种类似于 SQL 的语言来组织和查询数据，同时该类 SQL 语言也允许熟悉 MapReduce 的开发者开发自己定义的 Mapper 和 Reducer 来处理内建 Mapper 和 Reducer 无法完成的复杂分析任务。

Hive 起源自 Facebook，并由 Jeff Hammerbacher 领导的团队推动，在 Facebook 内部每天会搜集大量的数据，并需要在这些数据上进行大量分析。在 2006 年 Facebook 每天需要分析数十个 GB 左右的数据，在 2007 年增长到大约 TB 的量级。最初的分析是通过手工的 Python 脚本形式进行，后来将数据存储在 HDFS/HBase 中，并通过 MapReduce 程序进行分析。但是 MapReduce 是一个底层编程接口，对于数据分析人员来说需要进行大量的客户端编程及调试工作，另外用 HBase 作数据库没有类 SQL 的查询方式，操作和计算数据不方便，而且数据分析人员对 SQL 更加熟悉，于是 Hive 就诞生了。Hive 的设计目标是使 Hadoop 上的数据操作与传统 SQL 结合，让熟悉 SQL 编程的开发人员能够轻松向 Hadoop 平台转移。Hive 发展简史如表 8.1 所示。

表 8.1　Hive 发展历程

时　间	事　件
2008 年 8 月	Facebook 把 Hive 项目贡献给 Apache 基金会
2011 年 3 月	Hive 0.7.0 版本发布，此版本为重大升级版本，增加了简单索引、HAING 等众多高级特性
2011 年 6 月	Hive 0.7.1 版本发布，此版本修复了一些 Bug，如在 Windows 上使用 JDBC 问题
2011 年 12 月	Hive 0.8.0 版本发布，此版本为重大升级版本，加 Bitmap Indexes、TIMESTAMP datatype、Plugin Developer Kit、JDBC Driver Improvements 等新特性

续表

时　　间	事　　件
2012 年 2 月	Hive 0.8.1 版本发布，修复了一些 Bug，如使 Hive 可以同时运行在 Hadoop 0.20.x 与 0.23.0 上
2012 年 4 月	Hive 0.9.0 版本发布，重大改进版本，增加了对 Hadoop 1.0.0 的支持、实现 BETWEEN 等特性
2013 年 1 月	Hive 0.10.0 版本发布，重大改进版本，如支持创建 Cube 和 Rollup、更好地支持 YARN、联合查询优化等，并增加了对 Hadoop 2.x.y 的支持
2013 年 5 月	Hive 0.11.0 版本发布，新增 Explain dependency、Union 优化、实现了 TRUNCATE、建立了大量的关键字等
2013 年 10 月	Hive 0.12.0 版本发布，主要针对速度(Speed)、SQL、HCatalog 等方面的特性进行优化
2014 年 4 月	Hive 0.13.0 版本发布，采用了 ACID 语义的事务机制等
2014 年 6 月	Hive 0.13.1 版本发布，修复了一些 Bug，增加了一些新特性
2014 年 9 月	Hive 0.14.0 版本发布，修复了一些 Bug，增加了一些新特性
2015 年 2 月	Hive 1.0.0 版本发布，移除 HiveServer1，全面使用 Hive Server2；HiveMetaStoreClient 定义公开的 API 等
2015 年 3 月	Hive 1.1.0 版本发布，该版本提供了简单数据 ETL、利用各种数据格式结构的机制，通过 MapReduce 和 Tez 框架查询执行等
2015 年 5 月	Hive 1.2.0 版本发布，该版本进行了 Bug 修复、性能优化、增加了一些新特性等

由于 Hive 采用了类似 SQL 的查询语言 HiveQL，因此读者很容易将 Hive 理解为数据库。其实 Hive 为数据仓库设计，而传统的数据库可用在 Online 的应用中，只是这两者拥有类似的查询语言，具有同样的 SQL 界面而已。Hive 与数据库之间的具体差异如表 8.2 所示。

表 8.2　Hive 与数据库之间的差异

	数据仓库 Hive	关系型数据库 RDBMS
查询语句	HiveQL	SQL
数据存储位置	HDFS/HBase	Raw Device/Local FS
数据格式	用户定义	系统决定
数据更新	不支持(会把之前的数据覆盖)	支持
索引	0.8 版本之后增加，但弱	有
执行	MapReduce	Executor
执行延迟	高	低
可扩展性	高	低
数据规模	大(数据大于 TB)	小
数据检查	读时模式	写时模式

1. 查询语句

由于 SQL 已被广泛地应用在数据库操作中，因此 Hive 专门针对熟悉 SQL 的开发者提供了类 SQL 的查询语言 HiveQL，方便 SQL 开发者使用 Hive 进行开发。

2. 数据存储位置

传统的数据库是将数据保存在块设备 Raw Device 或本地文件系统 Local FS 上，而 Hive 是建立在 Hadoop 平台基础上，数据都是存储在 HDFS 或 HBase 中。

3. 数据格式

在传统数据库中，不同的数据库有不同的存储引擎，都已定义了自己的数据格式，存在数据库中所有的数据都会按照该数据格式进行存储。Hive 没有定义专门的数据格式，默认支持多种文件格式，同时也可以通过实现 MapReduce 的 InputFormat 或 OutputFormat 类由用户定制格式。

4. 数据更新

传统数据库主要对数据进行增、删、查、改等操作，支持数据更新。Hive 是针对数据仓库应用设计的，而数据仓库中的数据主要以读为主，因此，Hive 不支持对数据的更新操作。

5. 索引

传统数据库通常会针对一个或几个列建立索引，对于少量的特定条件的数据访问具有低延迟、高效率的特点。Hive 0.8 版本之前并没有索引，由于使用 MapReduce，Hive 可以并行访问数据，对于大数据量的访问，Hive 也可以体现出优势。在 Hive 0.8 版本之后，也增加了其索引，主要是为了降低访问延迟。

6. 执行

传统数据库都有自己的执行引擎。Hive 中大多数查询的执行都是通过 Hadoop 提供的 MapReduce 来实现的。

7. 执行延迟

传统数据库对于已建立索引的数据，其执行延迟较低，对于未建立索引的大规模数据，其执行延迟较高。Hive 一方面没有对数据建立索引，需要扫描整个表，因此执行延迟较高；另一方面 MapReduce 框架也是 Hive 执行延迟较高的原因之一。

8. 可扩展性

传统数据库是基于 ACID 语义的，因此扩展性非常有限。Hive 是建立在 Hadoop 集群之上的，因此 Hive 的可扩展性和 Hadoop 的可扩展性是一致的。

9. 数据规模

数据库设计本身存在缺陷,使得数据库可以支持的数据规模较小。Hive 是建立在 Hadoop 集群之上,并可以使用 MapReduce 进行计算,因此可以支持的数据规模比较大。

10. 数据检查

传统数据库的数据检查是写时模式(Schema on Write),即数据在写入数据库时对模式进行检查。这种写时模式有利于提升查询功能。Hive 的数据检查是读时模式(Schema on Read),即 Hive 在数据加载时不进行验证,而是在查询数据时才进行数据检查。在很多情况下,加载时模式是未知的,也不能决定使用何种索引时,适合使用读时模式。

另外,Hive 和 HBase 都是基于 Hadoop 架构之上,都是用 HDFS 作为底层存储,它们两个的主要区别在于 Hive 是建立在 Hadoop 之上为了减少 MapReduce Jobs 编写工作的批处理系统,HBase 是为了支持弥补 Hadoop 对实时操作缺陷的;Hive 本身不存储和计算数据,它完全依赖于 HDFS 和 MapReduce,Hive 中的表是纯逻辑表,HBase 不是逻辑表,而是物理表,提供一个超大的内存 Hash 表,搜索引擎通过它来存储索引和方便查询操作。

8.2　Hive 特点

Hive 最大的特点就是基于 Hadoop 平台可以自动适应节点数目和数据量的动态变化,即可扩展性,还具有 MapReduce 和用户定义的函数库相结合的可延展性,并且具有良好的容错性和低约束的数据输入格式等特点。Hive 特点的具体说明如下。

1. 可扩展性

传统的并行数据仓库可以很好地扩展到几十或者上百个节点的集群,并且达到接近线性的加速比,但如果需要扩展到上千或者更多节点时,传统的并行数据仓库则无法满足要求。而 Hive 与传统的并行数据仓库最大的不同之处在于 Hive 更加关注水平扩展性,即可通过简单地增加资源来支持更大的数据量和负载,其水平扩展性可达到上千以上。Hive 的水平可扩展性使得能够处理 PB 级的数据量。

2. 可延展性

Hive 的可延展性体现在支持用户自定义函数方面,用户可以根据自己的需求来实现自己的函数。如 Hive 定义的类 SQL 语言(HiveQL),允许用户进行和 SQL 相似的操作,而且还允许开发人员方便地开发自己定义的 Mapper 和 Reducer 来处理内建 Mapper 和 Reducer 无法完成的复杂分析任务。

3. 容错性

Hive 是基于 MapReduce 框架，Hive 的执行计划在 MapReduce 框架上以作业的形式执行，每个作业的中间结果文件写到本地存储，最终输出文件写到 HDFS 文件系统中，利用 HDFS 的多副本机制来保证数据的可靠性，从而达到作业的容错性，即 Hive 的容错性。当作业在执行过程中某节点出现故障，也不会影响到 Hive 的执行计划。

4. 低约束性

传统的并行数据仓库需要先把数据装载到数据库中，并按照特定的格式进行存储，然后才能执行查询操作。Hive 与传统并行数据仓库的不同之处在于 Hive 具有低约束性，即 Hive 没有自己专门的数据存储格式，也没有为数据建立索引(Hive 0.8 版本之前)，用户可以非常自由地组织 Hive 中的表。Hive 在执行查询前无须导入数据，直接执行查询操作，而且 Hive 默认支持多种文件格式，同时也可以通过实现 MapReduce 的 InputFormat 或 OutputFormat 类由用户定制格式。

8.3 Hive 体系架构

Hive 可被认为是一种数据仓库，并提供了完整的 SQL 查询功能，它建立在 Hadoop 的其他组件之上，依赖于 HDFS/HBase 进行数据存储，并依赖于 MapReduce 完成数据查询操作。Hive 可将结构化的数据文件映射为一张数据库表，可将 SQL 语句转换为 MapReduce 任务进行，并按照该计划生成 MapReduce 任务后交给 Hadoop 集群处理。Hive 的体系框架如图 8.1 所示。

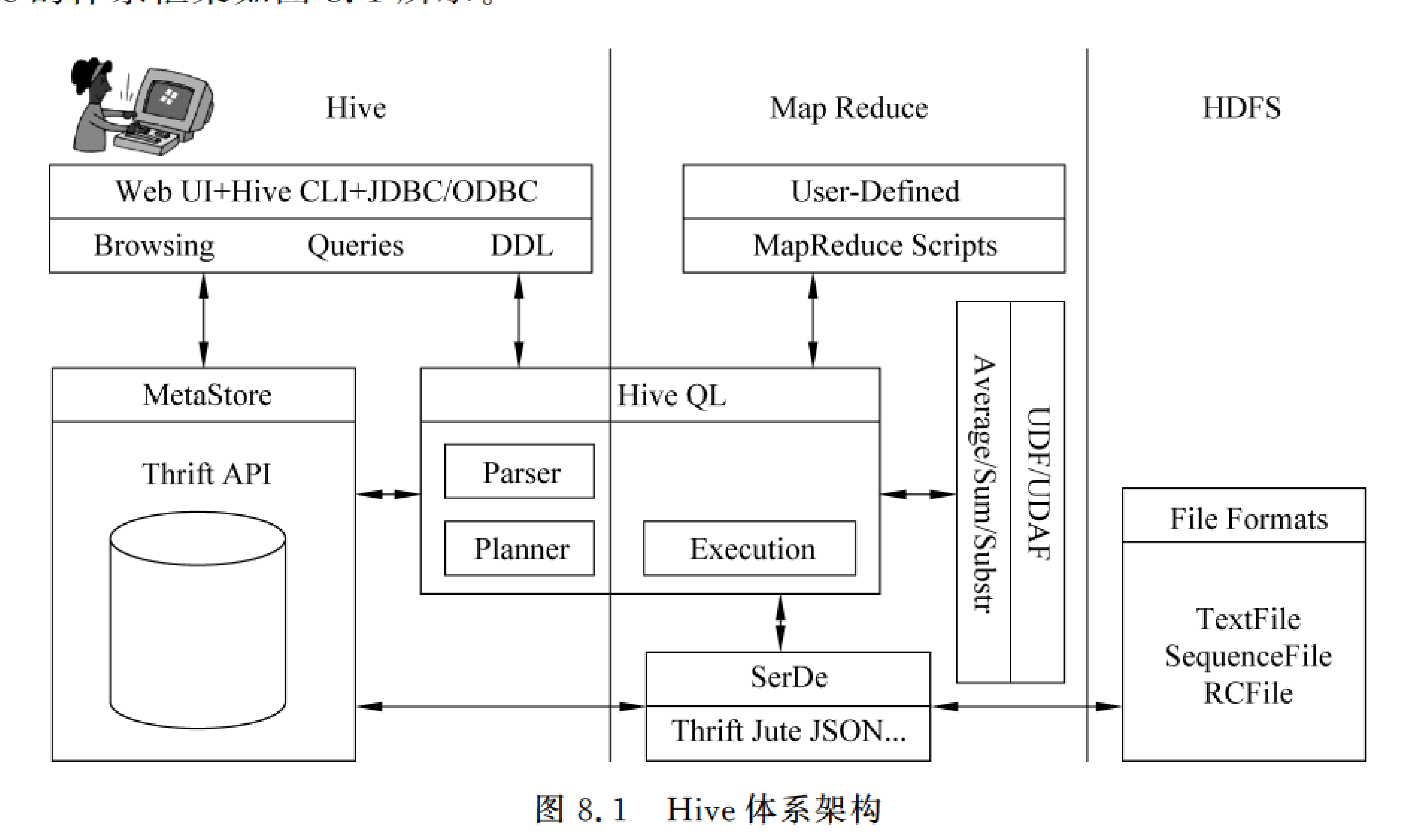


图 8.1 Hive 体系架构

从图 8.1 可以看出，Hive 的体系架构主要由用户接口、元数据存储 MetaStore、HiveQL、MapReduce、HDFS/HBase 存储组成，具体说明如下。

1. 用户接口

Hive 的用户接口主要用于提交查询和其他操作，包含类似命令行接口 CLI、Thrift Client、Web UI 和 JDBC/ODBC。其中，命令行接口 CLI 为 Shell 命令行；Web UI 是通过浏览器访问 Hive，该接口对应 Hive 的 hwi 组件（Hive Web Interface），使用前要启动 hwi 服务；JDBC/ODBC 是 Hive 的 Java，与使用传统数据库 JDBC 的方式类似；Thrift Client 是为其他 Client 端提供 Thrift 服务，如 JDBC 和 ODBC 接口。Thrift 服务简化了在多编程语言中运行 Hive 命令，Hive 的 Thrift 绑定支持 C++、Java、PHP、Python 和 Ruby 等。

2. 元数据存储 MetaStore

Hive 将元数据存储在数据库中，如 MySQL、Derby。Hive 中的元数据包括表的名字、列、分区及其属性、表的属性（是否为外部表等）、表的数据所在目录等。Hive 默认使用内存数据库 Derby 存储元数据，使用时不需要修改任何配置，缺点就是当 Hive 重启后所有的元数据都会丢失。Hive 还可使用 MySQL、Oracle 等任何支持 JDBC 连接方式的数据库来存储元数据，此时需要修改相应的配置项。

3. HiveQL

HiveQL 组件可看作驱动器 Driver，包括解释器 Parser、执行计划 Panner 和执行引擎。Hive 通过解释器 Parser 将 HiveQL 编译成中间表示，包括对 HiveQL 的分析，执行计划的生成以及优化等工作。最后由执行引擎 Execution 完成具体的执行操作，包括 MapReduce 执行、HDFS 操作、元数据操作等。

4. SerDe

SerDe 是 Serialize/Deserilize 的简称，主要用于序列化（将 Java 对象转换成存储格式）和反序列化（从存储格式转换成 Java 对象）。用户在 Hive 中建表时可以用自定义的 SerDe 或者使用 Hive 自带的 DerDe 为表指定列，并对列指定相应的数据。

5. Hadoop

Hadoop 包括 MapReduce 和 HDFS/HBase，这一部分其实并不包括在 Hive 之中，它是 Hive 存储和计算的实际载体。Hive 的数据存储在 HDFS/HBase 中，大部分的查询由 MapReduce 完成。

8.4　Hive 安装配置

要使得 Hive 在 Hadoop 集群中充分发挥作用，就需要相应的软件、硬件及以太网络的支撑。下面将针对 Hive 系统正常运行所需要的软硬件环境、网络环境、安装模式、配置、部署等方面进行介绍，帮助初学者对 Hive 进行更深入的研究和学习。

8.4.1 准备工作

Hive是Hadoop生态系统中的一个组件，在Hadoop集群规划时，就应该考虑到对Hive的集群规划(有关集群规划的相关内容请查看3.6节)。在没有实际业务需求，以研究和学习为目的，并帮助初学者快速搭建Hive集群环境的情况下，本实例将在表3.7的Hadoop集群搭建的基础上(软硬件环境及网络环境)对Hive集群中的各节点角色进行分配，具体节点角色分配和配置参数信息如表8.3所示。

表8.3 Hive集群节点角色分配

机器名称	角色	IP地址	硬件参数	操作系统
Master1.Hadoop	NameNode SecondaryNameNode ResourceManager HMaster MySQL	192.168.1.100	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave1.Hadoop	DataNode NodeManager HRegionServer ZooKeeper Hive(Server)	192.168.1.101	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave2.Hadoop	DataNode NodeManager HRegionServer ZooKeeper Hive(Client)	192.168.1.102	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7
Slave3.Hadoop	DataNode Nodemanager HRegionServer ZooKeeper Hive(Client)	192.168.1.103	内存 1GB 处理器 1 硬盘(SCSI) 20GB	CentOS 7

由表8.3可以看出，Master机器主要配置MySQL，主要负责元数据的存储；Slave机器配置Hive，主要负责HiveQL的解析、编译优化、生成执行计划、调用底层的MapReduce计算框架等。

8.4.2 安装模式

Hive的安装模式有内嵌模式、本地独立模式和远程模式三种。这三种安装模式的主要区别如图8.2所示。

1. 内嵌模式

内嵌模式(Embedded Metastore)是将元数据保持在内嵌的Derby数据库中，一个内嵌的Derby数据库每次只允许一个会话连接，不支持多会话连接，只适用于简单的测试。

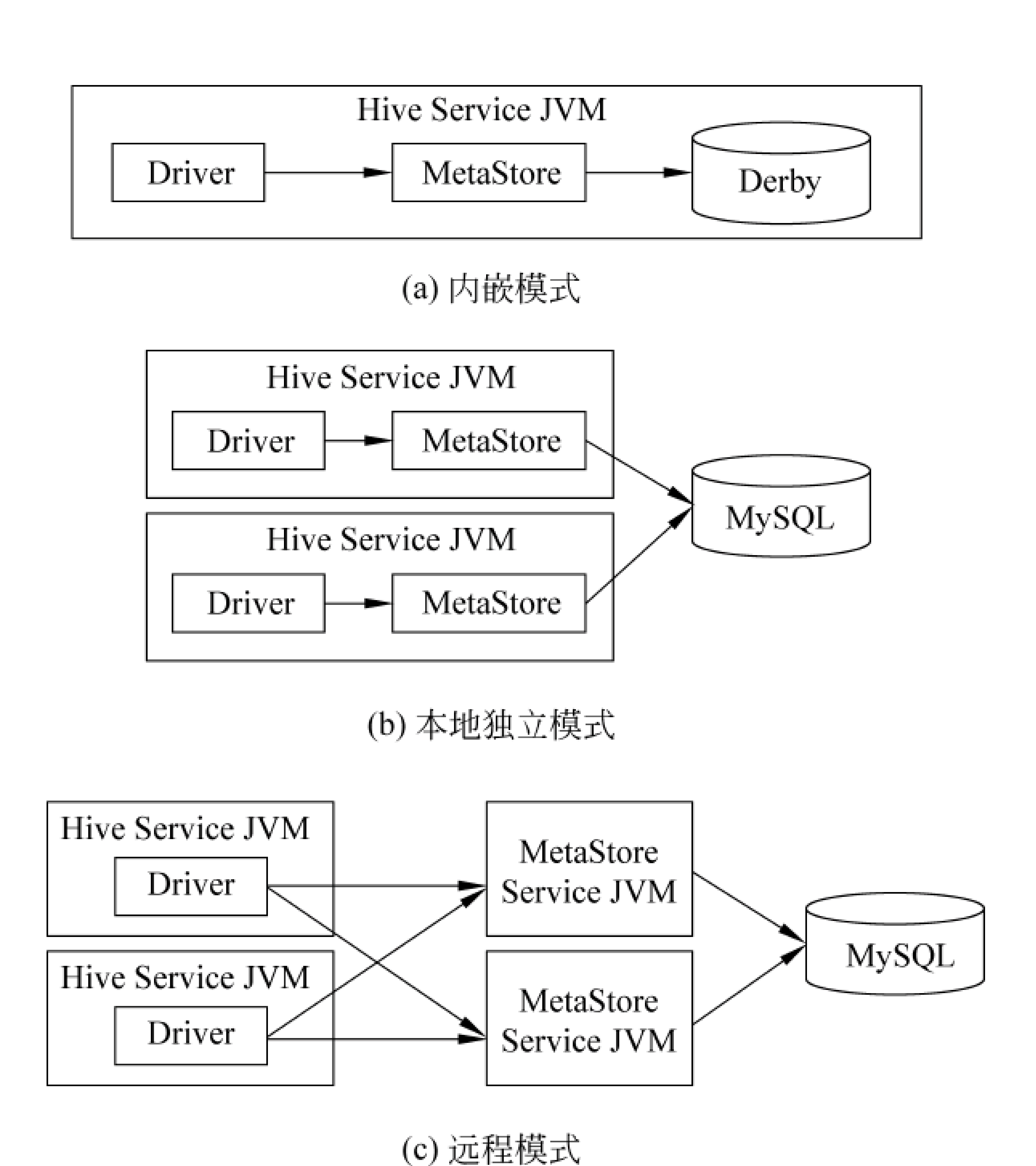


图 8.2　Hive 三种安装模式

2. 本地独立模式

本地独立模式(Local Metastore)是在本地安装数据库,把元数据放在本地数据库中,常用的是使用 MySQL 作为元数据库,这种模式能够支持多会话和多用户连接。

3. 远程模式

远程模式(Remote Metastore)是把元数据放置在远程的数据库,Hive 服务通过 Thrift 访问元数据 MetaStore,这种模式可以控制数据库的连接等。

8.4.3　安装 Hive

本实例将采用远程模式来安装 Hive,并使用 MySQL 作为元数据库。首先需要在原有 Hadoop 的集群节点中选择安装 MySQL 和 Hive 的节点(Hive 原则上可以安装在集群上的任何一个节点上),这里仅使用 Master1.Hadoop 节点作为 Hive 和 MySQL(用于存储 Hive 元数据)的节点。具体安装过程如下。

1. 安装 MySQL

MySQL 可通过 yum 命令在线下载安装、下载离线 rpm 安装包安装、下载源码编译安装三种方式进行安装。本实例采用下载离线 rpm 安装包的方式进行 MySQL 的安装。MySQL 到目前为止的最新版本为 MySQL Community Server 5.6.24,读者可从 MySQL

官网进行下载(http://dev.mysql.com/downloads/mysql/),本实例所使用的 MySQL 版本为 5.5.4(请查看本书配套资料中 Softwares 文件夹下的 MySQL-server-5.5.40-1.el6.x86_64.rpm)。在安装 MySQL 之前,首先需要确认系统是否已经安装相应的数据库。因为 Master1.Hadoop 节点安装了 CentOS 7 版本的 Linux 操作系统,默认安装了 mariadb-libs,必须先卸载才可以继续安装 MySQL,具体的操作命令如下。

```
[root@master1 hadoop]#rpm -qa|grep -i mariadb-libs
                                                    //查找是否安装 mariadb-libs
mariadb-libs-5.5.35-3.el7.x86_64
[root@master1 hadoop]#yum remove mariadb-libs.x86_64
                                                    //卸载 mariadb-libs
Loaded plugins: fastestmirror, langpacks
Repository base is listed more than once in the configuration
Repository updates is listed more than once in the configuration
Repository extras is listed more than once in the configuration
Repository centosplus is listed more than once in the configuration
Repodata is over 2 weeks old. Install yum-cron?Or run: yum makecache fast
...
Removed:
  mariadb-libs.x86_64 1:5.5.35-3.el7

Dependency Removed:
  postfix.x86_64 2:2.10.1-6.el7

Complete!
[root@master1 hadoop]#rpm -qa|grep -i mysql         //检查是否安装 MySQL
//安装 MySQL
[root@master1 hadoop]#rpm -ivh MySQL-server-5.5.40-1.el6.x86_64.rpm
Preparing...                          ###########################[100%]
Updating / installing...
   1:MySQL-server-5.5.40-1.el6        ###########################[100%]
...
[root@master1 bin]#service mysql start              //启动 MySQL
Starting MySQL.. SUCCESS!
[root@master1 bin]#service mysql status             //查看 MySQL运行状态
 SUCCESS! MySQL running(3281)
```

2. 配置 MySQL

安装好 MySQL 之后,就要对 MySQL 进行配置,如创建 Hive 元数据库、指定 hive 用户及授权等,来满足 Hive 的元数据存储要求,具体配置过程如下。

```
//MySQL安装完成后只有一个 root管理账号,但是此时的 root账号并没有密码
[hadoop@master1 ~]$mysql -u root -p                    //通过该命令来登录 MySQL数据库
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 261
Server version: 5.5.40 MySQL Community Server(GPL)
…
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

//在 MySQL中为 Hive创建 hive用户,数据库和授权
mysql>CREATE USER 'hive' IDENTIFIED BY 'hive';         //创建 hive用户
mysql>GRANT ALL PRIVILEGES ON * .*  TO 'hive'@'%' WITH GRANT OPTION;
                                                       //授权
mysql>flush privileges;
mysql>exit;                                            //退出 MySQL数据库
[hadoop@master1 ~]$mysql -uhive -phive                 //使用 hive账号登录 MySQL数据库
mysql>create database hive;                            //创建 hive数据库
mysql>show databases;                                  //显示已创建的数据库
+--------------------+
| Database           |
+--------------------+
| information_schema |
| hive               |
| performance_schema |
| test               |
+--------------------+
4 rows in set(0.00 sec)
```

至此,完成了在 MySQL 数据库中为 Hive 创建了元数据库(数据库名为 hive),指定 Hive 用户(用户名为 hive,密码为 hive)及授权等配置。

3. 安装 Hive

Hive 的安装过程与 Hadoop 和 HBase 的安装过程相似,先下载与 Hadoop 和 HBase 版本对应的 Hive 文件安装包(可从 Hive 官网 http://hive.apache.org/下载 Hive 的最新版本,到目前为止 Hive 的最新版为 1.2.0),然后将压缩包解压到集群节点的安装目录中。本实例的 Hadoop 版本为 2.6.0,HBase 版本为 1.0.0,与其匹配的 Hive 最新版本为 Hive 1.2.0。因此,在官网下载了 Hive 1.2.0 版本或在本书配套资料中的 Softwares 目录下查看 Hive(文件名为 apache-hive-1.2.0-bin.tar.gz)文件压缩包,然后将压缩包解压到 Slave1.Hadoop 节点的/opt/目录下,具体命令过程如下。

```
#tar -zxvf apache-hive-1.2.0-bin.tar.gz          //解压 Hive二进制文件
#mv apache-hive-1.2.0-bin /opt/                  //移动 Hive 1.2.0到"/opt/"目录下
```

```
//把"/opt/ apache-hive-1.2.0-bin"读权限分配给 hadoop 用户
#chown -R hadoop:hadoop /opt/ apache-hive-1.2.0-bin
#vi /etc/profile                              //把 Hive 安装路径添加到配置文件 profile 中
 export HIVE_HOME=/opt/apache-hive-1.2.0-bin
 export PATH=$PATH:$JAVA_HOME/bin:$HIVE_HOME/bin
#su hadoop                                    //切换为 hadoop 用户
```

通过上述过程，就完成了 Hive 二进制文件的解压、hadoop 权限、配置环境变量等过程。同理，可将集群规划中的 Slave2. Hadoop 和 Slave3. Hadoop 都进行上述操作或者将 Slave1. Hadoop 的 Hive 安装包复制到 Slave2. Hadoop 和 Slave3. Hadoop 的相应目录中。

8.4.4 配置 Hive

当 Hive 安装完成之后，就需要对 Hive 进行配置。Hive 使用和 Hadoop 类似的 XML 配置文件进行设置，配置文件在<HIVE_HOME>/conf 目录下，如 hive-env. sh、hive-site. xml 和 hive-log4j. properties 配置文件。

1. 配置 hive-env. sh

在启动 Hive 时，会使用到 hive-env. sh 配置文件，该配置文件的配置可参考<HIVE_HOME>/conf 目录下的 hive-env. sh. template，配置内容如下。

```
#Set HADOOP_HOME to point to a specific hadoop install directory
HADOOP_HOME=/opt/hadoop-2.6.0                   //Hadoop 安装目录
```

2. 配置 hive-site. xml

本实例的 Hive 在启动时会加载两个配置文件，一个是默认配置文件 hive-default. xml，另一个就是用户自定义文件 hive-site. xml。当 hive-site. xml 中的配置参数的值与 hive-default. xml 文件中不一致时，以用户自定义的 hive-site. xml 配置文件为准。因此，可使用 hive-default. xml 文件来保留默认配置，hive-site. xml 用于个性化配置，来覆盖默认配置，具体的操作如下。

```
[hadoop@slave1 ~]$ su root                      //切换为 root 用户
Password:
[root@slave1 hadoop]#cd /opt/apache-hive-1.2.0-bin/conf/
                                                //进入 Hive conf 目录
[root@slave1 conf]#cp hive-default.xml.template hive-default.xml
                                                //复制
[root@slave1 conf]#chown -R hadoop:hadoop hive-default.xml
                                                //授权 hadoop 用户
[root@slave1 conf]#cp hive-default.xml.template hive-site.xml
```




```
                                                   //复制
[root@ slave1 conf]#chown -R hadoop:hadoop hive-site.xml
                                                   //授权给 hadoop 用户
```

在 Hive 安装目录的 conf 目录下没有 hive-default. xml 和 hive-site. xml 文件，只有一个 hive-default. xml. template 文件，所以要复制该文件，并且分别命名为 hive-default. xml 和 hive-site. xml。由于是使用 root 用户操作，还需要把这两个文件授权给 hadoop 用户。然后，需要配置 Hive Server 端（Slave1. Hadoop 节点）的 hive-site. xml 文件，其配置内容如下。

```
[root@ slave1 conf]#vim hive-site.xml
<!—hive-site.xml 文件的配置-->
<property>
  <name>javax.jdo.option.ConnectionURL</name>
   <value>
   jdbc:mysql://master1.hadoop:3306/hive?createDatabaseIfNotExist=true
   </value>
  <description>JDBC connect string for a JDBC metastore</description>
</property>

<property>
  <name>javax.jdo.option.ConnectionDriverName</name>
  <value>com.mysql.jdbc.Driver</value>
  <description>Driver class name for a JDBC metastore</description>
</property>

<property>
  <name>javax.jdo.option.ConnectionUserName</name>
  <value>hive</value>
  <description>username to use against metastore database</description>
</property>

<property>
  <name>javax.jdo.option.ConnectionPassword</name>
  <value>hive</value>
  <description>password to use against metastore database</description>
</property>

<property>
  <name>hive.metastore.warehouse.dir</name>
  <!--base hdfs path -->
  <value>/user/hive/warehouse</value>
</property>
```

其中，javax. jdo. option. ConnectionURL 用来配置 Metastore 数据库的 JDBC URL，本实例 Hive 的 Metastore 元数据库为 Master1. Hadoop 节点上 MySQL 数据库中的 hive 元数据库；javax. jdo. option. ConnectionDriverName 用来配置 JDBC 驱动器类名，即 com. mysql. jdbc. Driver；javax. jdo. option. ConnectionUserName 用来配置 JDBC 用户名，在配置 MySQL 数据库时指定了 hive 用户名；javax. jdo. option. ConnectionPassword 用来配置 JDBC 密码，在配置 MySQL 数据时指定了 hive 密码；hive. metastore. warehouse. dir 参数用来配置表的存储位置，这里使用了默认值/user/hive/warehouse。如果读者想了解有关 hive-site. xml 的更多信息，请查看＜HIVE_HOME＞/conf 目录下的 hive-default. xml. template 文件或查看附表 4 的内容。

由于本实例 Hive 的安装模式为远程模式，因此还需要对 Hive 客户端(Slave2. Hadoop 节点和 Slave3. Hadoop 节点)进行配置，对 Hive Client 端的 hive-site. xml 文件配置内容如下。

```
<!--thrift://<host_name>:<port>默认端口是 9083 -->
<property>
  <name>hive.metastore.uris</name>
  <value>thrift://Slave1.Hadoop:9083</value>
  <description>Thrift uri for the remote metastore. Used by metastore client to connect to remote
metastore.</description>
</property>

<!--hive表的默认存储路径 -->
<property>
  <name>hive.metastore.warehouse.dir</name>
  <value>/user/hive/warehouse</value>
  <description>location of default database for the warehouse</description>
</property>
```

其中，hive. metastore. uris 用于指定 Hive 元数据访问路径，该属性一般都配置在 Hive Client 端，其格式如下。

```
<property>
     <name>hive.metastore.uris</name>
     <value>uri1,uri2,...</value>                  //可配置多个 uri
    <description>JDBC connect string for a JDBC metastore</description>
</property>
```

URI 的格式为 thrift://Hive Server 端 IP：端口号(默认端口号为 9083)。thrift 是 Hive 的通信协议，用于指定 IP 的节点上启动 Hive 服务，而且 value 值可以配置多个 URI，表示可以访问多个 Hive Server 端；hive. metastore. warehouse. dir 用于指定 Hive 的数据存储目录，默认值是/user/hive/warehouse。

3. 使用 JDBC 连接元数据

安装并配置完 Hive 之后，需要复制 MySQL 的 JDBC 驱动包到<HIVE_HOME>/lib 目录下，这样可通过 JDBC 连接到元数据库。本实例 MySQL JDBC 的驱动包所使用的版本为 mysql-connector-java-5.1.16.jar（本书配套资料的 Softwares 目录下的 mysql-connector-java-5.1.16.jar 文件），将其复制到 Hive 的 lib 目录下。

8.4.5 启动 Hive

当完成上述的 Hive 安装与配置之后，就可以通过 hadoop 用户权限启动 Hive。由于本实例采用的是 Hive 远程模式，在启动时需要先启动 Hive Server 端的 Metastore 服务，再由 Hive Client 端通过 Hive Shell 连接访问。Hive Server 端的 Metastore 服务启动命令为：

```
hive --service metastore -p <port_num>
```

如果不加端口默认启动：hive --service metastore，则默认监听端口是 9083。这里需要注意的是，Client 端中的端口配置需要和启动监听的端口一致。Server 端启动正常后，Client 端就可以执行 hive 操作了。Hive 具体启动命令如下。

```
//Hive Server 端 Slave1.Hadoop 的 Hive Metastore 服务启动
[hadoop@slave1 ~]$ hive --service metastore
Starting Hive Metastore Server

//Hive Client 端 Slave2.Hadoop 启动 Hive Shell
[hadoop@slave2 ~]$ hive
Logging initialized using configuration in file:/opt/apache-hive-1.2.0-bin/conf/hive-log4j.properties
hive>

//Hive Client 端 Slave2.Hadoop 启动 Hive Shell
[hadoop@slave3 ~]$ hive
Logging initialized using configuration in file:/opt/apache-hive-1.2.0-bin/conf/hive-log4j.properties
hive> create table test(key string);                //创建一个 test 表,测试其可用性
OK
Time taken: 9.308 seconds
hive> show tables;                                  //显示已创建的表
OK
test
Time taken: 0.19 seconds, Fetched: 1 row(s)
hive>
```

通过上述命令，成功启动了 Hive Server 端的 Hive Metastore 服务和 Hive Client 端的 Hive Shell，然后又在 Slave3. Hadoop 节点的 Hive Client 端创建了一个名为 test 的表，并通过 show tables 命令显示已创建的表。在 Hive Shell 中，如果一个命令或者查询执行成功了，则先输出 OK，然后才会紧跟着输出内容，最后一行表示命令或者查询执行所消耗的时间的输出信息结尾。在 Hive Client 端创建的 test 表已通过 Hive Server 端存储在了 MySQL 数据库中，具体的查看命令如下。

```
[hadoop@master1 ~]$ mysql -uhive -phive
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 299
Server version: 5.5.40 MySQL Community Server (GPL)
...
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> use hive;                                        //使用 Hive 数据库
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;                                     //显示 Hive 数据库存中的元数据表
+---------------------------+
| Tables_in_hive            |
+---------------------------+
| BUCKETING_COLS            |
| CDS                       |
| COLUMNS_V2                |
| DATABASE_PARAMS           |
| DBS                       |
| FUNCS                     |
| FUNC_RU                   |
| GLOBAL_PRIVS              |
| PARTITIONS                |
| PARTITION_KEYS            |
| PARTITION_KEY_VALS        |
| PARTITION_PARAMS          |
| PART_COL_STATS            |
| ROLES                     |
| SDS                       |
| SD_PARAMS                 |
| SEQUENCE_TABLE            |
| SERDES                    |
| SERDE_PARAMS              |
```

```
| SKEWED_COL_NAMES             |
| SKEWED_COL_VALUE_LOC_MAP     |
| SKEWED_STRING_LIST           |
| SKEWED_STRING_LIST_VALUES    |
| SKEWED_VALUES                |
| SORT_COLS                    |
| TABLE_PARAMS                 |
| TAB_COL_STATS                |
| TBLS                         |
| VERSION                      |
+------------------------------+
29 rows in set (0.00 sec)
```

通过 show tables 命令，读者可看到在 MySQL 数据库中的 Hive 元数据表，如 TBLS 表示所有 Hive 表的基本信息；TABLE_PARAMS 表示表级属性；SERDE_PARAMS 表示序列化反序列化信息；PARTITIONS 表示 Hive 表分区信息；PARTITION_KEYS 表示 Hive 分区表分区键；PARTITION_KEY_VALS 表示 Hive 表分区名等。例如，可通过 mysql> select * from TBLS;命令查看 Hive 的元数据信息(如图 8.3 所示)。

```
+--------+-------------+-------+------------------+--------+-----------+-------+----------+---------------+--------------------+--------------------+
| TBL_ID | CREATE_TIME | DB_ID | LAST_ACCESS_TIME | OWNER  | RETENTION | SD_ID | TBL_NAME | TBL_TYPE      | VIEW_EXPANDED_TEXT | VIEW_ORIGINAL_TEXT |
+--------+-------------+-------+------------------+--------+-----------+-------+----------+---------------+--------------------+--------------------+
|      1 |  1433386069 |     1 |                0 | hadoop |         0 |     1 | test     | MANAGED_TABLE | NULL               | NULL               |
+--------+-------------+-------+------------------+--------+-----------+-------+----------+---------------+--------------------+--------------------+
1 row in set (0.00 sec)
```

图 8.3　Hive 的元数据信息

通过上述的一系列步骤，完整地演示了如何安装和配置 Hive，并通过添加一张数据库表的演示，来说明 Hive 和 MySQL 元数据库之间的关系操作。请读者深刻体会和理解。

8.5　Hive 数据模型

Hive 是基于 Hadoop 分布式文件系统的，本身并没有专门的数据存储格式，也没有为数据建立索引，只需要在创建表时告诉 Hive 数据中的列分隔符和行分隔符就可以解析数据。Hive 的数据主要分为表数据和元数据，表数据是 Hive 中表格(Table)具有的数据；元数据是用来存储表的名字，表的列、分区及其属性等。因此，Hive 的数据模型主要由表构成，包括内部表(Table)、外部表(External Table)、分区表(Partition Table)、桶表(Bucket Table)。

1. 内部表

Hive 中的内部表和关系型数据库中的表的概念是相似的，能被过滤、投影、连接和合并。表的创建过程和数据加载过程可以在同一个语句中完成，当删除表时，表中的数据和元数据将一同被删除。每个 Hive 的内部表在 HDFS 中都有相应的目录用来存储表中的

数据，该目录可通过 ${HIVE_HOME}/conf/hive-site.xml 配置文件中的 hive.metastore.warehouse.dir 属性进行配置，该属性的默认值是/user/hive/warehouse(这个目录在 HDFS 上)，用户可以根据实际情况来修改该配置。

2. 外部表

Hive 中的外部表和关系型数据库中的表概念很相似，也能被过滤、投影、连接和合并。与 Hive 中内部表的主要区别在于外部表只是一个过程，即表的创建和加载是同时完成的。外部表中真正的数据不是放在自己表所属的目录中，而是存储在(CREATE EXTERNAL TABLE … LOCATION)LOCATION 之后指定的 HDFS 路径中。当删除外部表时，并不删除实际的数据，而只是删除相应的元数据。

3. 分区表

Hive 中的分区表是指在创建表时指定的 Partition 的分区空间，即把数据粗粒度地划分成块。Hive 引入分区表的目的是可以让查询发生在小范围的数据上，避免扫描整个表内容，从而提高了数据查询效率。如果需要创建有分区的表，需要在 create 表时调用可选参数 partitioned by，具体的语法结构如下所示。

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
[(col_name data_type [COMMENT col_comment], ...)]
[COMMENT table_comment]
[PARTITIONED BY(col_name data_type [COMMENT col_comment], ...)]
```

一个表可以有一个或多个分区；每个分区以文件夹的形式单独存在表文件夹的目录下；分区是以字段的形式在表结构中存在，通过 describe table 命令可以查看到字段，但是该字段不存放实际的数据内容，仅仅是分区的表示。

4. 桶表

对于每个表或者分区，Hive 可以进一步组织成桶表(Bucket Table)，桶是更为细粒度的数据范围划分。表和分区者是基于目录级别的拆分数据，而桶则是对数据源数据文件本身来拆分数据，使用桶的表会将源数据文件按一定规律拆分成多个文件。Hive 引入桶表的目的是为了获得更高的查询处理效率，它能使一些特定的查询效率更高，如对于具有相同的桶划分并且 Join 的列刚好就是在桶里的连接查询等。如果需要创建有桶的表或分区，需要在 create 表时调用可选参数 clustered by (id) into 4 buckets，具体的语法结构如下所示。

```
CREATE TABLE bucketed_user(id INT)name STRING)
CLUSTERED BY(id)INTO 4 BUCKETS;
```

关键字 clustered 声明划分桶的列和桶的个数，这里以用户的 ID 来划分 4 个桶。Hive 采用对列值哈希，然后除以桶的个数求余的方式决定该条记录存放在哪个桶当中。

8.6　Hive 数据类型

Hive 支持很多关系型数据库都支持的基本数据类型(不同长度的整型、浮点型、无长度限制的字符串类型和布尔类型等),同时还支持关系型数据库中很少出现的 4 种复杂数据类型(数组 ARRAY、映射 MAP、结构体 STRUCT 和联合体 UNION)。

8.6.1　基本数据类型

Hive 支持多种不同长度的整型、浮点型、布尔类型、无长度限制的字符串类型等。由于 Hive 是由 Java 语言编写的,Hive 的基本数据类型和 Java 的基本数据类型也是一一对应的(字符串类型除外,Hive 的字符串类型相当于数据库的 varchar 类型,该类型是一个可变的字符串),如数据类型 TINYINT、SMALLINT、INT 和 BIGINT 分别等价于 Java 语言中的 byte、short、int 和 long 数据类型;Hive 的浮点数据类型 FLOAT 和 DOUBLE 分别对应 Java 语言中的 float 和 double 基本数据类型;Hive 中的布尔类型 BOOLEAN 相当于 Java 的基本数据类型 boolean。表 8.4 列出了 Hive 所支持的基础数据类型。

表 8.4　Hive 基本数据类型

数据类型		描　述
Numeric	TINYINT	1 字节(8 位)有符号整数
	SMALLINT	2 字节(16 位)有符号整数
	INT	4 字节(32 位)有符号整数
	BIGINT	8 字节(64 位)有符号整数
	FLOAT	4 字节(32 位)单精度浮点数
	DOUBLE	8 字节(64 位)双精度浮点数
	DECIMAL	可以自定义精度进行扩展(Hive 0.11.0 以上支持)
Misc	BOOLEAN	true/false
	BINARY	字节数组(Hive 0.8 以上支持)
Date/Time	TIMESTAMP	整数、浮点数或字符串(Hive 0.8 以上支持)
	DATE	YYYY-MM-DD,Date 类型支持的范围是 0000-01-01～9999-12-31,依赖于原始的 Java 支持的日期类型(Hive 0.12.0 以上支持)
String	STRING	字符串(可使用单引号或双引号)
	CHAR	字符类型,使用单引号(Hive 0.13.0 以上支持)
	VARCHAR	创建的长度是 1～65 535,在字符串中容许使用最大数量的字符(Hive 0.12.0 以上支持)

从表 8.4 可以看出,Hive 支持的基本数据类型包括 TINYINT、SMALLINT、INT、BIGINT、FLOAT、DOUBLE、BOOLEAN、STRING、TIMESTAMP 和 BINARY。其中,

TIMESTAMP 和 BINARY 数据类型是 Hive 0.8 版本以上新增加的数据类型；DECIMAL 数据类型是 Hive 0.11.0 版本以上新增加的数据类型；VARCHAR 和 DATE 数据类型是 Hive 0.12.0 版本以上新增加的数据类型；CHAR 数据类型是 Hive 0.13.0 版本以上新增加的数据类型。

8.6.2 复杂数据类型

Hive 的复杂数据类型包括 ARRAY、MAP、STRUCT 和 UNION，这些复杂数据类型都是由基础类型组成的。其中，ARRAY 类型是由一系列相同数据类型的元素组成，这些元素可通过下标来访问(下标从 0 开始)；MAP 类型是由 Key→Value 键值对组成，可通过 Key 来访问元素；STRUCT 可以包含不同数据类型的元素，这些元素可以通过"变量.元素"的方式来得到所需要的元素；UNION 类型是联合多种数据类型的结果集，并合并为一个独立的结果集，该数据类型是从 Hive 0.7.0 版本以后开始支持的。表 8.5 列出了 Hive 所支持的复杂数据类型。

表 8.5 Hive 复杂数据类型

数据类型		描述	示例
Complex	ARRAY	类似于 C 语言的结构体，其域可用点号访问	ARRAY<data_type>
	MAP	Key→Value 对，其域通过 Key 进行访问	MAP<primitive_type, data_type>
	STRUCT	同数据类型的有序序列，以 0 开始的整数进行索引	STRUCT<col_name : data_type [COMMENT col_comment], ...>
	UNION	可以综合上面的数据类型组合到一起	UNIONTYPE<data_type, data_type, ...>

大部分关系型数据库并不支持这些复杂数据类型，因为使用这些复杂数据类型可能会破坏标准格式，如在传统关系型数据库中，STRUCT 数据类型可能需要多个不同的表拼装，而表间需要适当地使用外键来进行连接。这里通过一个具体的实例来说明如何使用这些数据类型，虚构一张学生信息表如下所示。

```
CREATE TABLE students(
  name      STRING,
  sex       CHAR,
  num       INT,
  info      ARRAY<STRING>,
  course    MAP<STRING, FLOAT>,
  address   STRUCT<street:STRING, city:STRING, state:STRING, zip:INT>);
```

在 Hive 中新创建了一个名为 students 的表，其中，name(学生姓名)使用字符串 STRING 来表示；sex(性别)使用字符 CHAR 来表示；num(学号)使用整型 INT 来表示；info(学生信息)列表是一个字符串值数组，该数组中的每一个元素都将会引用这张表中

的另一条记录；course（课程信息）是一个由 Key-Value 对构成的 MAP，其记录了每门课程所对应的成绩；address（学生地址）使用 STRUCT 数据类型，对每个域都做了命名（street、city、state 和 zip），并且每个域都指定了一个特定的类型（STRING、STRING、STRING 和 INT）。

8.6.3　数据类型转换

Hive 同 Java 语言一样，Hive 数据类型之间也可以相互转换。Hive 包括隐式转换（Implicit Conversions）和显式转换（Explicitly Conversions）。任何整数类型都可以隐式转换成一个范围更大的类型，如低字节的基本类型可以转化为高字节的类型，如 TINYINT、SMALLINT、INT 可以转化为 FLOAT 类型，而所有的整型类型、FLOAT 类型及 STRING 类型都可以转化为 DOUBLE 类型。表 8.6 列出了 Hive 支持的数据类型之间是否可以进行隐式的转换操作。

表 8.6　Hive 支持的数据类型是否可进行隐式转换

	BL	TY	SI	INT	BI	FL	DB	DM	ST	VC	TS	DATE	BA
VOID	T	T	T	T	T	T	T	T	T	T	T	T	T
BL	T	F	F	F	F	F	F	F	F	F	F	F	F
TY	F	T	T	T	T	T	T	T	T	T	F	F	F
SI	F	F	T	T	T	T	T	T	T	T	F	F	F
INT	F	F	F	T	T	T	T	T	T	T	F	F	F
BI	F	F	F	F	T	T	T	T	T	T	F	F	F
FL	F	F	F	F	F	T	T	T	T	T	F	F	F
DB	F	F	F	F	F	F	T	T	T	T	F	F	F
DM	F	F	F	F	F	F	F	T	T	T	F	F	F
ST	F	F	F	F	F	F	T	T	T	T	F	F	F
VC	F	F	F	F	F	F	T	T	T	T	F	F	F
TS	F	F	F	F	F	F	F	F	T	T	T	F	F
DATE	F	F	F	F	F	F	F	F	T	T	F	T	F
BA	F	F	F	F	F	F	F	F	F	F	F	F	T

注：由于表格过大，这里对一些较长的数据类型进行缩写。其中，BL 是 BOOLEAN 的缩写；TY 是 TINYINT 的缩写；SI 是 SMALLINT 的缩写；BI 是 BIGINT 的缩写；FL 是 FLOAT 的缩写；DB 是 DOUBLE 的缩写；DM 是 DECIMAL 的缩写；ST 是 STRING 的缩写；VC 是 VARCHAR 的缩写；TS 是 TIMESTAMP 的缩写；BA 是 BINARY 的缩写。

表 8.6 中的 T 是 TRUE 的缩写，表示可进行隐式转换，F 是 FALSE 的缩写，表示不

可进行隐式转换。若读者想了解两种数据类型之间是否可隐式转换，可查看表 8.6 的内容，如 BOOLEAN 类型和 BINARY 类型不能隐式转换为其他任何类型，DATE 类型和 TIMESTAMP 类型可以隐式转换为 STRING 类型和 VARCHAR 类型。

Hive 也支持显式类型转换，如将高字节类型转化为低字节类型，这就需要使用 Hive 的自定义函数 CAST 来实现，CAST 的语法结构如下。

```
cast(<expr> AS <TYPE>)
```

使用 CAST 时需要注意以下几点。

(1) 如果要将 FLOAT 型的数据转换成 INT 型数据，内部操作是通过 round()或者 floor()函数来实现的，而不是通过 CAST 实现。

(2) 对于 BINARY 类型的数据，只能将 BINARY 类型的数据转换成 STRING 类型，如果 BINARY 类型数据是一个数字类型，可利用嵌套的 CAST 操作来转化，比如 a 是一个 BINARY 类型，且是一个数字类型，则可使用下面的方式进行显式转化。

```
cast(cast(a as STRING)as INT)
```

(3) 对于 DATE 类型的数据，只能在 DATE、TIMESTAMP 及 STRING 类型之间进行转换，具体转换方式如下所示。

```
cast(DATE as DATE)              //将 DATE 类型转化为 DATE 类型
cast(TIMESTAMP as DATE)         //将 TIMESTAMP 类型转化为 DATE 类型
cast(STRING as DATE)            //将 STRING 类型转化为 DATE 类型
                                //如果 STRING 不是 YYYY-MM-DD 格式，结果会返回 NULL
cast(DATE as TIMESTAMP)         //将 DATE 类型转化为 TIMESTAMP 类型
cast(DATE as STRING)            //将 DATE 类型转化为 YYYY-MM-DD 的 STRING 类型
```

(4) 如果数据类型转换不成功，则将返回的是 NULL。

8.7 Hive 基本操作

Hive 的官方文档对 Hive 的基本操作有比较详细的描述(有兴趣的读者可参考 Hive 官方网址查看 http://wiki.apache.org/hadoop/Hive /LanguageManual)。Hive 的基本操作大体可以分为 DDL(Data Definition Language，数据定义语言)操作、DML(Data Manipulation Language，数据操作语言)操作和 Hive Shell 操作三种类型。

8.7.1 DDL 操作

DDL 操作主要用于数据表(内表、外部表、分区表、桶表)或数据库的创建、删除、修改和显示等操作，具体的操作指令如下。

```
• CREATE DATABASE/SCHEMA, TABLE, VIEW, FUNCTION, INDEX
• DROP DATABASE/SCHEMA, TABLE, VIEW, INDEX
• TRUNCATE TABLE
• ALTER DATABASE/SCHEMA, TABLE, VIEW
• MSCK REPAIR TABLE (or ALTER TABLE RECOVER PARTITIONS)
• SHOW DATABASES/SCHEMAS, TABLES, TBLPROPERTIES, PARTITIONS, FUNCTIONS,
  INDEX[ES], COLUMNS, CREATE TABLE
• DESCRIBE DATABASE/SCHEMA, table_name, view_name
```

读者在使用时，要根据每条指令的语法进行操作，如创建一个指定名字的表的语法格式如下。

```
CREATE [EXTERNAL] TABLE [IF NOT EXISTS] table_name
    [(col_name data_type [COMMENT col_comment], ...)]
    [COMMENT table_comment]
    [PARTITIONED BY (col_name data_type [COMMENT col_comment], ...)]
    [CLUSTERED BY (col_name, col_name, ...)
    [SORTED BY (col_name [ASC|DESC], ...)] INTO num_buckets BUCKETS]
    [ROW FORMAT row_format]
    [STORED AS file_format]
    [LOCATION hdfs_path]
```

有关创建表的语法格式说明如下。

(1) CREATE TABLE 创建一个指定名字的表，如果有相同名称的表存在，则抛出异常；也可使用 IF NOT EXISTS 选项来忽略该异常；

(2) EXTERNAL 关键字用来创建外部表，在建表时指定一个指向实际数据的路径(LOCATION)；

(3) PARTITIONED BY 语句用于创建有分区的表，一个表中可以有一个或多个分区，每个分区单独在一个目录下；

(4) CLUSTERED BY 语句用于创建桶表，可以对表和分区将若干个列放入一个桶中；

(5) SORTED BY 语句用于对数据进行排序，这样可以为特定应用提高性能；

(6) COMMENT 语句用于对表和列进行注释；

(7) STORED AS 语句用于指定数据存储格式，如果要指定数据存储格式为纯文本，可以使用 STORED AS TEXTFILE，如果数据需要压缩，可使用 STORED AS SEQUENCE。

例如，创建一个名为 test_table 的普通表：

```
CREATE TABLE test_table    #表名
(
   id      INT,                                      #字段名称    字段类型
   name    STRING,
```

```
    no      INT
   )COMMENT '注释:XXX'                          #表注释
   PARTITIONED BY(pt STRING)                    #分区表字段
ROW FORMAT DELIMITED
    FIELDS TERMINATED BY '\001'                 #指定字段分割符
STORED AS SEQUENCEFILE;                         #指定数据的存储方式
                                                #SEQUENCEFILE 是 Hadoop 自带的文件压缩格式
SHOW TABLES;                                    #查看所有的表
SHOW TABLES '*test*'                            #模糊查询含 test 关键字的表
SHOW PARTITIONS TABLES;                         #查看表有哪些分区
DESCRIBE TABLE;                                 #查看表结构
```

创建带有 BUCKET 的 test_bucket 表：

```
CREATE TABLE test_bucket
(
   id      INT,
   name   STRING,
   no      INT
)
PARTITIONED BY(pt STRING)
CLUSTERED BY(id) INTO 3 BUCKETS
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t'
STORED AS TEXTFILE;
```

创建 EXTERNAL 表：

```
CREATE EXTERNAL TABLE test_external
(
    id     INT,
    name STRING,
    no    INT
)
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','
STORED AS TEXTFILE
LOCATION '/user/data/test_hive.txt';
```

创建与已知表相同结构的表(只复制表结构,不复制表内容)：

```
CREATE TABLE test_like_table LIKE test_bucket;
```

另外,SHOW DATABASES 语句用于列出前当数据库,还可以使用 LIKE 从句利用正则表达式对数据库进行过滤(通配符只能是“*”或者“|”),具体的语法结构为：

```
SHOW(DATABASES|SCHEMAS)[LIKE identifier_with_wildcards];
```

SHOW TABLES 语句列出当前数据库中所有的表和视图，若使用 IN 从句则列出指定数据库中的所有表和视图，还可以使用正则表达式进行过滤，具体的语法结构为：

```
SHOW TABLES [IN database_name] [identifier_with_wildcards];
```

SHOW PARTITIONS 语句实现以字母顺序列出指定表中的所有分区，具体的语法结构为：

```
SHOW PARTITIONS table_name  [PARTITION(partition_desc)]
```

SHOW INDEXES 语句用于输出特定表上的所有索引信息，包括索引名称、表名、被索引的列名、保存索引的表名、索引类型和注释等，具体的语法结构为：

```
SHOW [FORMATTED](INDEX|INDEXES)ON table_with_index [(FROM|IN)db_name]
```

SHOW COLUMNS 语句用于输出给定表中包含分区列的所有列，具体的语法结构为：

```
SHOW COLUMNS(FROM|IN)table_name [(FROM|IN)db_name]
```

SHOW FUNCTIONS 语句用于输出匹配正则表达式的自定义和内置的函数（使用“. * ”输出所有函数），具体的语法结构为：

```
SHOW FUNCTIONS"a.* "
```

SHOW LOCKS 语句用于显示表或者分区上的锁，具体的语法结构为：

```
SHOW LOCKS<table_name>;
SHOW LOCKS<table_name>EXTENDED;
SHOW LOCKS<table_name>PARTITION(<partition_desc>);
SHOW LOCKS<table_name>PARTITION(<partition_desc>)EXTENDED;
```

SHOW TRANSACTIONS 语句用于查询当前打开或者终止的事务，具体的语法结构为：

```
SHOW TRANSACTIONS
```

SHOW COMPACTIONS 语句用于显示当前 Hive 事务被使用时，所有正在被压缩或预定压缩的表和分区，具体的语法结构为：

```
SHOW COMPACTIONS
```

DESCRIBE DATABASE 语句用于显示给定数据库的注释、在 HDFS 上的路径和数据库的拥有者信息，具体的语法结构为：

```
DESCRIBE DATABASE db_name
```

DESCRIBE TABLE/VIEW/COLUMN 语句用于显示给定表包括分区列在内的所有列，如果使用了 EXTENDED 关键字，则以 Thrift 序列化形式显示表的元数据；如果使用 FORMATTED 关键字，则以表格形式显示元数据；如果表拥有复合类型的列，则通过“表名.复合列名”的形式查看该列的属性，具体的语法结构为：

```
DESCRIBE [EXTENDED|FORMATTED] [db_name.]table_name[DOT col_name([DOT field_name] | [DOT '$elem$'] |
[DOT '$key$'] | [DOT '$value$'])* ]
#'$elem$'用于数组,'$key$'用于 map 的键,'$value$'用于 map 的键值
```

DESCRIBE PARTITION 语句用于显示指定分区列的元数据，具体的语法结构为：

```
DESCRIBE [EXTENDED|FORMATTED] [db_name.]table_name PARTITION partition_spec
```

DROP TABLE 用于删除一个内部表的同时会删除表的元数据和数据，删除一个外部表，只删除元数据而保留数据。ALTER TABLE 语句允许用户改变现有表的结构，用户可以增加列/分区，改变 Serialize/Deserilize、增加表、表本身重命名等。

8.7.2 DML 操作

DML 操作主要用于数据的加载、添加、查询等操作。本节将重点介绍加载数据 LOAD、添加数据 INSERT 和查询数据 SELECT 这三种操作。

1. LOAD

LOAD 操作只是单纯的复制/移动操作，即将数据文件移动到 Hive 表对应的位置，具体的语法结构如下。

```
#LOAD
LOAD DATA [LOCAL] INPATH 'filepath' [OVERWRITE] INTO
TABLE tablename [PARTITION(partcol1=val1, partcol2=val2 ...)]
```

有关 LOAD 的语法格式说明如下。

(1) 如果指定了 LOCAL 关键字，则 LOAD 命令会去查找本地文件系统中的 filepath；如果没有指定 LOCAL 关键字，filepath 若给定一个完整的 URL，Hive 会直接使用该 URL，否则会使用 Hadoop 配置文件中定义的 schema 和 authority。

(2) filepath 用于指定数据路径，可以指定相对路径、绝对路径、包含模式的完整 URL。

(3) 如果指定了 OVERWRITE 关键字，则目标表/分区中的内容会被删除，然后再将

filepath 指向的文件/目录中的内容添加到表/分区中；如果目标表/分区已有一个文件，且文件名和 filepath 中的文件名冲突，则现有的文件会被新文件所替代。

例如，为前面所创建的表 test_table（假定该表所在的目录为/usr/hadoop/hive）加载数据（数据文件路径为/usr/hadoop/data/20150708.txt），加载数据命令如下。

```
LOAD DATA LOCAL INPATH '/usr/hadoop/data/20150708.txt'
OVERWRITE INTO TABLE test_table PARTITION(pt='20150708');
```

该命令表示从本地磁盘把文件"/usr/hadoop/data/20150708.txt"复制到表 test_table 分区 pt 为"20150708"的目录下。当"/usr/hadoop/data/20150708.txt"文件加载成功后，会放置在 hdfs://master1.hadoop:900/usr/Hadoop/hive/test_table/pt=20150708/20150708.txt。

2. INSERT

INSERT 操作是将查询结果插入到 Hive 表中，支持基本插入模式、多插入模式、动态分区模式等，Hive 0.8 版本以上新增了 INSERT INTO。

1）基本插入模式

INSERT 的基本插入模式的语法结构如下。

```
#基本插入模式
INSERT OVERWRITE TABLE tablename1 [PARTITION(partcol1=val1, partcol2=val2 ...) [IF NOT EXISTS]] select
_statement1 FROM from_statement;
```

INSERT OVERWRITE 语句将会覆盖表或者分区中任何存在的数据，但如果为分区指定了 IF NOT EXISTS 关键字则不会覆盖分区中的数据。

2）多插入模式

INSERT 的多插入模式最小化了要求的数据扫描次数，Hive 可以仅扫描输入数据一次并应用不同的查询操作符，然后将数据插入多个表中，其语法结构如下。

```
#多插入模式
FROM from_statement
INSERT OVERWRITE TABLE tablename1 [PARTITION(partcol1=val1, partcol2=val2 ...)] select_statement1
[INSERT OVERWRITE TABLE tablename2 [PARTITION ...] select_statement2] ...
```

3）动态分区模式

在 INSERT 的动态分区模式中，用户可以在 PARTITION 从句中仅指定分区列的名称，分区列的值是可选的。如果给定分区列值，则该分区列为静态分区，否则为动态分区。每个动态分区列对应 select 子句的一个输入列，动态分区列必须在 select 子句中所有输入列的最后指定，并且与它们的 PARTITION 从句出现的顺序保持一致，其语法结构如下。

```
#动态分区模式
INSERT OVERWRITE TABLE tablename PARTITION (partcol1 [= val1], partcol2 [= val2] ...) select_statement FROM
from_statement
```

在默认情况下动态分区模式是禁用的，需要修改 hive. error. on. empty. partition（在动态分区插入产生空结果时，是否抛出异常）、hive. exec. dynamic. partition（true/false，是否允许动态分区插入）、hive. exec. dynamic. partition. mode（strict/nostrict，strict 模式中用户至少指定一个静态分区以防用户覆盖所有分区，nostrict 模式中所有分区都允许是动态的）等配置参数来实现动态分区插入。

4）INSERT INTO

INSERT INTO 语句用于添加数据到表或者分区中，并且保持现存数据的完整性，在插入数据时必须通过指定所有分区列的值来指定表的分区，其语法结构如下。

```
#INSERT INTO
INSERT INTO TABLE tablename1 [PARTITION (partcol1= val1, partcol2= val2 ...)] select_statement1 FROM
from_statement
```

3. SELECT

Hive 中的 SELECT 语句与传统 SQL 语句中的 SELECT 有些不同，该语句可以作为 UNION 查询一个子语句或者另一个查询的子查询，具体的语法结构为：

```
[WITH CommonTableExpression (, CommonTableExpression) * ]
SELECT [ALL | DISTINCT]select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING where_condition]
[ORDER BY col_list]
[CLUSTER BY col_list
  | [DISTRIBUTE BY col_list] [SORT BY col_list]
]
[LIMIT number]
```

有关 SELECT 语句的语法格式说明如下。

（1）COMMON TABLE EXPRESSION 是由 WITH 子句指定的简单查询得到的临时结果集，仅在一条语句的执行范围内定义；

（2）WHERE where_condition 从句是布尔表达式，只有满足此表达式的记录才会返回，从 Hive 0.13 版本开始，一些子查询可以出现在 WHERE 从句中，这些子查询的结果被作为 IN 和 NOT IN 语句中的常量；

（3）ALL｜DISTINCT 语句用于区分对重复记录的处理，默认为 ALL，表示查询所

有记录，DISTINCT 表示去掉重复的记录；

(4) 当使用 GROUP BY col_list 从句时，SELECT 语句只能包含那些包含在 GROUP BY 从句中的列，或者在 SELECT 语句中使用聚类函数 COUNT/SUM 等；

(5) ORDER BY col_list 从句与 SQL 中的 ORDER BY 相似，用于对输入做全局排序，都支持 ASC 和 DESC；

(6) SORT BY col_list 从句与 ORDER BY 语法格式相似，不是全局排序，在数据进入 Reducer 之前，对某个 Reducer 进行排序，如果存在多个 Reducer，SORT BY 只保证每个 Reducer 的输出有序，不保证全局有序；

(7) CLUSTER BY 和 DISTRIBUTE BY 从句与 Map/Reduce 脚本一起使用，DISTRIBUTE BY 将数据分到同一个 Reducer，CLUSTER BY 是 DISTRIBUTE BY 和 SORT BY 的简捷方式，除了具有 DISTRIBUTE BY 的功能外，还会对该字段进行排序；

(8) LIMIT number 语句可以限制查询的记录数，其中查询的结果是随机选择的，如果需要返回前几条记录，可先对结果进行排序再使用 LIMIT 从句。

8.8　Hive 内置运算符

Hive 有 4 种类型的运算符，即关系运算符、算术运算符、逻辑运算符和复杂运算符。本节将介绍每种类型运算符的含义和基本用法。

8.8.1　关系运算符

关系运算符被用来比较两个操作数。Hive 中可用的关系运算符如表 8.7 所示。

表 8.7　Hive 可用的关系运算符

运算符	操　作	描　述
A=B	所有基本类型	如果表达式 A 等于表达式 B，结果为 TRUE，否则为 FALSE
A!=B	所有基本类型	如果表达式 A 不等于表达式 B，返回 TRUE，否则返回 FALSE
A<B	所有基本类型	TRUE，如果表达式 A 小于表达式 B；否则 FALSE
A<=B	所有基本类型	TRUE，如果表达式 A 小于或等于表达式 B；否则 FALSE
A>B	所有基本类型	TRUE，如果表达式 A 大于表达式 B；否则 FALSE
A>=B	所有基本类型	TRUE，如果表达式 A 大于或等于表达式 B；否则 FALSE
A IS NULL	所有类型	TRUE，如果表达式的计算结果为 NULL；否则 FALSE
A IS NOT NULL	所有类型	FALSE，如果表达式 A 的计算结果为 NULL；否则 TRUE
A LIKE B	字符串	TRUE，如果字符串模式 A 匹配到 B；否则 FALSE
A RLIKE B	字符串	NULL，如果 A 或 B 为 NULL；TRUE，如果 A 任何子字符串匹配 Java 正则表达式 B；否则 FALSE
A REGEXP B	字符串	等同于

假设 Hive 数据库中有一张名为 student 的表,该表由字段 num、name、dept、score 组成,数据内容如下。

```
+-----+--------------+--------+--------------------+
| num | name        | dept | score   |
+-----+--------------+------------------------------+
|1001 | jack          | CS   |   80   |
|1002 | mills         | CS   |   85   |
|1003 | vera          | CS   |   90   |
|1004 | michelle      | CS   |   95   |
|1005 | alice         | CS   |   100   |
+-----+--------------+--------+--------------------+
```

当查询学号为 1001 学生的详细信息时,可通过下面的查询命令实现。

```
hive> SELECT * FROM student WHERE num=1001;
+-----+--------------+--------+--------------------+
| num | name        | dept | score   |
+-----+--------------+------------------------------+
|1001 | jack          | CS   |   80   |
+-----+--------------+--------+--------------------+
```

当查询学生成绩大于 90 分的学生信息时,可使用下面的查询命令实现。

```
hive> SELECT * FROM student WHERE score >=90;
+-----+--------------+--------+--------------------+
| num | name        | dept | score   |
+-----+--------------+------------------------------+
|1003 | vera          | CS   |   90   |
|1004 | michelle      | CS   |   95   |
|1005 | alice         | CS   |   100   |
+-----+--------------+--------+--------------------+
```

8.8.2 算术运算符

算术运算符用于两个操作数之间的常见算术运算,其返回类型为数字类型。Hive 中可用的算术运算符如表 8.8 所示。

表 8.8 Hive 可用的算术运算符

运算符	操　作	描　述	示　例
A+B	所有数字类型	A 加 B 的结果	hive> select 1+2 from test;
A−B	所有数字类型	A 减去 B 的结果	hive> select 1−2 from test;

续表

运算符	操　　作	描　　述	示　　例
A * B	所有数字类型	A 乘以 B 的结果	hive> select 1 * 2 from test;
A/B	所有数字类型	A 除以 B 的结果	hive> select 1/2 from test;
A%B	所有数字类型	A 除以 B. 产生的余数	hive> select 1%2 from test;
A&B	所有数字类型	A 和 B 的按位与结果	hive> select 1&2 from test;
A|B	所有数字类型	A 和 B 的按位或结果	hive> select 1|2 from test;
A^B	所有数字类型	A 和 B 的按位异或结果	hive> select 1 ^ 2 from test;
～A	所有数字类型	A 按位非的结果	hive> select ～2 from test;

8.8.3　逻辑运算符

逻辑运算符用于两个操作数之间的逻辑表达，其返回类型为 TRUE 或者 FALSE。Hive 中可用的逻辑运算符如表 8.9 所示。

表 8.9　Hive 可用的逻辑运算符

运 算 符	操　作	描　　述
A AND B	boolean	TRUE，如果 A 和 B 都是 TRUE；否则 FALSE
A && B	boolean	类似于 A AND B
A OR B	boolean	TRUE，如果 A 或 B 或两者都是 TRUE；否则 FALSE
A || B	boolean	类似于 A OR B
NOT A	boolean	TRUE，如果 A 是 FALSE；否则 FALSE
!A	boolean	类似于 NOT A

例如，查询 student 表中学生成绩在 90 分以上的计算机系（CS）学生的详细信息，可使用下面的查询命令实现。

```
hive> SELECT * FROM student WHERE score >= 90 && dept=CS;
+------+--------------+---------+--------------------+
| num | name       | dept | score  |
+------+--------------+----------------------------+
|1003 | vera        | CS  |  90   |
|1004 | michelle     | CS  |  95   |
|1005 | alice        | CS  |  100  |
+------+--------------+---------+--------------------+
```

8.8.4 复杂运算符

复杂运算符用于提供一个表达式来接入复杂类型的元素。Hive 中可用的复杂运算符如表 8.10 所示。

表 8.10 Hive 可用的复杂运算符

运算符	操　作	描　述
A[n]	A 是一个数组,n 是一个 int	返回数组 A 的第 n 个元素,第一个元素的索引为 0
M[key]	M 是一个 Map<K, V>,key 的类型为 K	返回对应于映射中关键字的值
S. x	S 是一个结构	返回 S 的 x 字段

8.9 Hive 内置函数

Hive 中可用的内置函数与 SQL 语句中的函数比较相似,可分为数值计算函数、日期函数、条件函数、字符串函数、集合统计函数、复杂类型长度统计函数等。

8.9.1 数值计算函数

Hive 数值计算函数有取整函数、取随机数函数、对数函数、幂运算函数等。Hive 中可用的数值计算函数如表 8.11 所示。

表 8.11 Hive 可用的数值计算函数

语　法	返回值	说　明
round(double a)	BIGINT	取整函数:返回 double 类型的整数值部分(遵循四舍五入)
round(double a, int d)	DOUBLE	指定精度取整函数:返回指定精度 d 的 double 类型
floor(double a)	BIGINT	向下取整函数:返回等于或者小于该 double 变量的最大的整数
ceil(double a)	BIGINT	向上取整函数:返回等于或者大于该 double 变量的最小的整数
rand()rand(int seed)	DOUBLE	取随机数函数:返回一个 0~1 范围内的随机数。如果指定种子 seed,则会得到一个稳定的随机数序列
exp(double a)	DOUBLE	自然指数函数:返回自然对数 e 的 a 次方
log10(double a)	DOUBLE	以 10 为底对数函数:返回以 10 为底的 a 的对数
log2(double a)	DOUBLE	以 2 为底对数函数:返回以 2 为底的 a 的对数
log(double base, double a)	DOUBLE	对数函数:返回以 base 为底的 a 的对数
pow(double a, double p)	DOUBLE	幂运算函数:返回 a 的 p 次幂

续表

语　　法	返回值	说　　明
power(double a，double p)	DOUBLE	幂运算函数：返回 a 的 p 次幂，与 pow 功能相同
sqrt(double a)	DOUBLE	开平方函数：返回 a 的平方根
bin(BIGINT a)	STRING	二进制函数：返回 a 的二进制代码表示
hex(BIGINT a)	STRING	十六进制函数：如果变量是 INT 类型，那么返回 a 的十六进制表示；如果变量是 STRING 类型，则返回该字符串的十六进制表示
unhex(string a)	STRING	反转十六进制函数：返回该十六进制字符串所代表的字符串
conv (BIGINT num， int from_base，int to_base)	STRING	进制转换函数：将数值 num 从 from_base 进制转化到 to_base 进制
abs(double a) abs(int a)	DOUBLE INT	绝对值函数：返回数值 a 的绝对值
pmod (int a， int b) pmod (double a，double b)	INT DOUBLE	正取余函数：返回正的 a 除以 b 的余数
sin(double a)	DOUBLE	正弦函数：返回 a 的正弦值
asin(double a)	DOUBLE	反正弦函数：返回 a 的反正弦值
cos(double a)	DOUBLE	余弦函数：返回 a 的余弦值
acos(double a)	DOUBLE	反余弦函数：返回 a 的反余弦值
positive (int a) positive (double a)	INT DOUBLE	positive 函数：返回 a
negative (int a) negative (double a)	INT DOUBLE	negative 函数：返回－a

8.9.2　日期函数

Hive 日期函数有 UNIX 时间戳转日期函数、日期时间转日期函数、日期转年/月/小时/分/秒函数、日期比较函数等。Hive 中可用的日期函数如表 8.12 所示。

表 8.12　Hive 可用的日期函数

语　　法	返回值	说　　明
from _ unixtime (bigint unixtime[，string format])	STRING	UNIX 时间戳转日期函数：转化 UNIX 时间戳(从 1970-01-01 00:00:00 UTC 到指定时间的秒数)到当前时区的时间格式
unix_timestamp()	BIGINT	获取当前 UNIX 时间戳函数：获得当前时区的 UNIX 时间戳
unix _ timestamp (string date)	BIGINT	日期转 UNIX 时间戳函数：转换格式为"yyyy-MM-dd HH:mm:ss"的日期到 UNIX 时间戳，如果转化失败，则返回 0

续表

语　法	返回值	说　明
unix_timestamp（string date，string pattern）	BIGINT	指定格式日期转 UNIX 时间戳函数：转换 pattern 格式的日期到 UNIX 时间戳，如果转化失败，则返回 0
to_date(string timestamp)	STRING	日期时间转日期函数：返回日期时间字段中的日期部分
year(string date)	INT	日期转年函数：返回日期中的年
month (string date)	INT	日期转月函数：返回日期中的月份
day (string date)	INT	日期转天函数：返回日期中的天
hour (string date)	INT	日期转小时函数：返回日期中的小时
minute (string date)	INT	日期转分钟函数：返回日期中的分钟
second (string date)	INT	日期转秒函数：返回日期中的秒
weekofyear (string date)	INT	日期转周函数：返回日期在当前的周数
datediff (string enddate，string startdate)	INT	日期比较函数：返回结束日期减去开始日期的天数
date_add (string startdate，int days)	STRING	日期增加函数：返回开始日期 startdate 增加 days 天后的日期
date_sub (string startdate，int days)	STRING	日期减少函数：返回开始日期 startdate 减少 days 天后的日期

8.9.3 条件函数

Hive 条件函数有 IF 函数、非空查找函数、条件判断函数等。Hive 中可用的条件函数如表 8.13 所示。

表 8.13 Hive 可用的条件函数

语　法	返回值	说　明
if (boolean testCondition，T valueTrue，T valueFalseOrNull)	T	If 函数：当条件 testCondition 为 TRUE 时，返回 valueTrue；否则返回 valueFalseOrNull
COALESCE(T v1，T v2，…)	T	非空查找函数：返回参数中的第一个非空值；如果所有值都为 NULL，那么返回 NULL
CASE a WHEN b THEN c [WHEN d THEN e] * [ELSE f] END	T	条件判断函数：如果 a 等于 b，那么返回 c；如果 a 等于 d，那么返回 e；否则返回 f

8.9.4 字符串函数

Hive 字符串函数有字符串长度函数、字符串反转函数、字符串连接函数、带分隔符串连接函数等。Hive 中可用的字符串函数如表 8.14 所示。

表 8.14　Hive 可用的字符串函数

语　　法	返回值	说　　明
length(string A)	INT	字符串长度函数：返回字符串 A 的长度
reverse(string A)	STRING	字符串反转函数：返回字符串 A 的反转结果
concat(string A, string B…)	STRING	字符串连接函数：返回输入字符串连接后的结果，支持任意个输入字符串
concat_ws(string SEP, string A, string B…)	STRING	带分隔符字符串连接函数：返回输入字符串连接后的结果，SEP 表示各个字符串间的分隔符
substr (string A, int start) substring(string A, int start)	STRING	字符串截取函数：返回字符串 A 从 start 位置到结尾的字符串
substr (string A, int start, int len) substring (string A, int start, int len)	STRING	字符串截取函数：返回字符串 A 从 start 位置开始，长度为 len 的字符串
upper(string A)ucase(string A)	STRING	字符串转大写函数：返回字符串 A 的大写格式
lower(string A) case(string A)	STRING	字符串转小写函数：返回字符串 A 的小写格式
trim(string A)	STRING	去空格函数：去除字符串两边的空格
ltrim(string A)	STRING	左边去空格函数：去除字符串左边的空格
rtrim(string A)	STRING	右边去空格函数：去除字符串右边的空格
regexp_replace(string A, string B, string C)	STRING	正则表达式替换函数：将字符串 A 中的符合 Java 正则表达式 B 的部分替换为 C。注意，在有些情况下要使用转义字符，类似 Oracle 中的 regexp_replace 函数
regexp _ extract (string subject, string pattern, int index)	STRING	正则表达式解析函数：将字符串 subject 按照 pattern 正则表达式的规则拆分，返回 index 指定的字符
parse_url(string urlString, string partToExtract [, string keyToExtract])	STRING	URL 解析函数：返回 URL 中指定的部分。partToExtract 的有效值为：HOST, PATH, QUERY, REF, PROTOCOL, AUTHORITY, FILE, and USERINFO
get _ json _ object (string json _ string, string path)	STRING	JSON 解析函数：解析 JSON 的字符串 json_string，返回 path 指定的内容。如果输入的 JSON 字符串无效，那么返回 NULL
space(int n)	STRING	空格字符串函数：返回长度为 n 的字符串
repeat(string str, int n)	STRING	重复字符串函数：返回重复 n 次后的 str 字符串
split(string str, string pat)	ARRAY	分割字符串函数：按照 pat 字符串分割 str，会返回分割后的字符串数组

8.9.5　集合统计函数

Hive 集合统计函数有个数统计函数、总和统计函数、平均统计函数、最小值统计函数、最大值统计函数、非空集合总体变量函数等。Hive 中可用的集合统计函数如表 8.15 所示。

表 8.15 Hive 可用的集合统计函数

语　　法	返　回　值	说　　明
count (*) count (expr) count (DISTINCT expr[, expr_.])	INT	个数统计函数：count(*)统计检索出的行的个数，包括 NULL 值的行；count(expr)返回指定字段的非空值的个数；count(DISTINCT expr[, expr_.])返回指定字段的不同的非空值的个数
sum(col)sum(DISTINCT col)	DOUBLE	总和统计函数：sum(col)统计结果集中 col 的相加的结果；sum(DISTINCT col)统计结果中 col 不同值相加的结果
avg(col)avg(DISTINCT col)	DOUBLE	平均值统计函数：avg(col)统计结果集中 col 的平均值；avg(DISTINCT col)统计结果中 col 不同值相加的平均值
min(col)	DOUBLE	最小值统计函数：统计结果集中 col 字段的最小值
max(col)	DOUBLE	最大值统计函数：统计结果集中 col 字段的最大值
var_pop(col)	DOUBLE	非空集合总体变量函数：统计结果集中 col 非空集合的总体变量(忽略 null)
var_samp (col)	DOUBLE	非空集合样本变量函数：统计结果集中 col 非空集合的样本变量(忽略 null)
stddev_pop(col)	DOUBLE	总体标准偏离函数：该函数计算总体标准偏离，并返回总体变量的平方根，其返回值与 VAR_POP 函数的平方根相同
stddev_samp (col)	DOUBLE	样本标准偏离函数：该函数计算样本标准偏离
percentile(BIGINT col, p)	DOUBLE	中位数函数：求准确的第 p 个百分位数，p 必须介于 0 和 1 之间，但是 col 字段目前只支持整数，不支持浮点数类型
percentile_approx(DOUBLE col, p [, B])	DOUBLE	近似中位数函数：求近似的第 p 个百分位数，p 必须介于 0 和 1 之间，返回类型为 double，但是 col 字段支持浮点类型，参数 B 控制内存消耗的近似精度，B 越大，结果的准确度越高
histogram_numeric(col, b)	ARRAY＜STRUCT {'x','y'}＞	直方图：以 b 为基准计算 col 的直方图信息

8.10 Hive 实例

在 Hadoop 生态圈中，Hive 实质上是一个 SQL 解析引擎，Hive 可以把 SQL 查询转换为 MapReduce 中的 Job 来运行。这里以 WordCount 项目为例(请读者查看 5.4.2 节中的 WordCount 类)，通过 MapReduce 的方式需要写很多 Java 代码，但是如果使用 Hive

就比较简单,具体的做法如下。

```
hive> create database db;                    #创建一个名为 db 的数据库
#在 db 数据库下创建一个名为 src_data 的表,存储路径为:/db/wordcount/src_data
hive> create external table src_data(line string) row format delimited fields terminated by '\n' stored
as textfile location '/db/wordcount/src_data'
create table words(word string);             #根据 MapReduce 的规则,对每行数据拆分成单词
insert into table words select explode(split(line, " "))as word from src_data;
select word, count(*)from db.words group by word;
                                             #实现单词统计
```

在该实例中,首先创建了一个名为"db"的数据库,在该数据库下创建了一个名为"src_data"的数据表(用于存储导入进来的文件,字段可以自己设置,表明存储什么内容),字段名为 text,类型为 string。表 src_data 中的数据可以从本地导入,也可以将 HDFS 里的数据导入到该表中,本实例是将本地的数据导入到 src_data 数据表中。数据导入成功之后,要使用 MapReduce 规则来进行单词统计的话,还需要创建一张表,用于存储分割字符串后的单词,这里创建一个名为 words 的表。然后编写 Hive 的 HQL,其中 split 是拆分函数(与 Java 中的 split 功能相同,按照空格拆分),Hive 将 HQL 解析成 MapReduce 任务,这样就基本实现了单词统计功能。最后通过 group by 实现单词统计功能。

请读者认真对比通过 MapReduce 和通过 Hive 的 HQL 这两种方式实现的 WordCount 的不同之处,可以明显地感觉到使用 Hive 进行操作更加简单、易用。Hive 是针对批量长时间数据分析设计的,不能做到交互式的实时查询,对于一些复杂的机器学习算法、复杂的科学计算的场景并不适用,而是适用于海量数据处理、数据挖掘、数据分析等应用场景。

第9章 数据分析与挖掘 Mahout

数据分析与挖掘是从海量的、不完整的、有噪声的数据中发现隐含在其中有价值的、潜在有用知识的前提。目前开源的数据分析与挖掘工具比较多，但适用于大数据分析与挖掘的并不多。其中，数据分析与挖掘框架 Mahout 是基于 Hadoop 实现的，把很多传统的数据挖掘算法转化为 MapReduce 的实现方式，大大提升了算法可处理的数据量和处理性能。本章将以 Mahout 为例来介绍数据分析与挖掘的相关知识。

9.1　Mahout 概述

Mahout 是机器学习和数据挖掘的一个分布式框架，是基于 Hadoop 的 MapReduce 实现了部分数据挖掘算法，将很多以前运行于单机上的数据挖掘算法转化为 MapReduce 模式的实现方式，从而大大提高了数据处理的性能。Mahout 最初起源于 2008 年，当时只是 Apache Lucene 的一个子项目，主要是为了解决文本搜索中的分类和聚类问题而被提出的。随后 Mahout 吸收了一个名为 Taste 的协同过滤开源项目，并逐渐发展成为具有较为完备的机器学习能力。在 2009 年时，发布了 Mahout Release 0.1 和 Release 0.2 版本；在 2010 年时，发布了 Mahout Release 0.3 和 Release 0.4 版本；在 2010 年 4 月时，Mahout 成为 Apache 的顶级项目；随后 Mahout 加入了对 Hadoop 的支持之后，Mahout 的核心目标是利用 Hadoop 的并行计算与处理能力实现可扩展的分布式机器学习框架。目前 Mahout 的最新版本为 Mahout 0.10(2015.4.11)，该版本已经可以支持一些传统的数据挖掘任务，即聚类、分类、推荐过滤、频繁子项挖掘等。

Mahout 是具有可扩充能力的机器学习类库，在提供了机器学习框架的同时，还实现了一些可扩展的机器学习领域经典算法，供开发人员在 Apache 的许可下免费使用，帮助开发人员更加方便快捷地创建智能应用程序。Mahout 经常使用于对海量数据处理和分析的情况，通过 Mahout 库所提供的算法构建在 MapReduce 框架之上，将算法的输入、中间结果和输出结果构建于 HDFS 分布式文件系统之上，从而使得 Mahout 具有高吞吐量、高并发性和高可靠性的特点。另外，Mahout 框架具有良好的扩展性和容错性、文档化好、实例丰富、完全源代码开放、技术社区活跃、易于使用等特点，已被广泛应用于推荐引擎、聚类、分类、相关性分析等应用场景。

注：如果读者想对 Mahout 有更深入的了解，建议认真阅读文章 *Map-Reduce for Machine Learning on Multicore*[86]，此后又并入了更多广泛的机器学习方法。

9.2　Mahout 安装配置

Mahout 是运行在 Hadoop 集群之上的机器学习算法库，是用 Java 语言编写完成的。在安装 Mahout 之前，需要保证 Hadoop 集群服务能够正常运行。Mahout 的安装过程比较简单，具体的安装步骤如下。

9.2.1　Mahout 安装

Mahout 需要 JDK 1.6 版本或更高版本的支持，由于 Mahout 的安装是在前面已搭建好的 Hadoop 集群上进行的，因此非常简单，只需在 Mahout 官网（http://mahout.apache.org/）下载与 Hadoop 版本相应的 Mahout 版本（本实例下载的为 Mahout 0.10.0 版本，并将 Mahout 安装在 Hadoop 集群的 Master1.Hadoop 节点上），下载完成后解压到 Master1.Hadoop 节点的/opt/目录下完成 Mahout 的安装，具体命令如下。

```
#tar -zxvf mahout-distribution-0.10.0.tar.gz      --解压 Hadoop 安装文件
#chown -R hadoop:hadoop /opt/mahout-distribution-0.10.0
```

本实例使用的为 Mahout 0.10 版本，每个版本都有 Mahout-distribution-version-src（Maven 工具管理维护的源代码，面向于研究和拓展算法）和 Mahout-distribution-version（可导入算法集 jar 包，面向于直接应用）两种类型的发布文本。本实例使用 Mahout-distribution-version 版本。

9.2.2　Mahout 配置

Mahout 安装成功之后，下一步将要配置三个环境变量：MAHOUT_HOME、PATH 和 CLASSPATH，三个变量的设置如表 9.1 所示。

表 9.1　三个变量的设置

变量名	变量值
MAHOUT_HOME	指明 Mahout 安装路径，此路径下包括 lib，bin，examples，docs，conf 等文件夹
PATH	使得系统可以在任何路径下识别 Mahout 命令
CLASSPATH	CLASSPATH 为 Mahout 加载类路径，只有类在 CLASSPATH 中，Mahout 命令才能识别

修改/etc/profile 文件，将 MAHOUT_HOME、PATH 和 CLASSPATH 添加到 profile 配置文件中，具体执行命令如下。

```
#vi /etc/profile                                  --把 Hadoop 安装路径添加到配置文件 profile 中
 export MAHOUT_HOME=/opt/mahout-distribution-0.10.0/
 export PATH=$PATH:$MAHOUT_HOME/bin
```

```
 export CLASSPATH=$CLASSPATH:$MAHOUT_HOME/lib
#source /etc/profile                              --立即生效
#su hadoop                                        --切换为 hadoop 用户
$start-all.sh                                     --启动 Hadoop 集群
$mahout --help                                    --查看 Mahout 目前拥有的算法
Running on hadoop, using /opt/hadoop-2.6.0/bin/hadoop and HADOOP_CONF_DIR=
MAHOUT-JOB: /opt/mahout-distribution-0.10.0/mahout-examples-0.10.0-job.jar
Valid program names are:
  arff.vector: : Generate Vectors from an ARFF file or directory
  baumwelch: : Baum-Welch algorithm for unsupervised HMM training
  buildforest: : Build the random forest classifier
  canopy: : Canopy clustering
  cat: : Print a file or resource as the logistic regression models would see it
  cleansvd: : Cleanup and verification of SVD output
  clusterdump: : Dump cluster output to text
  clusterpp: : Groups Clustering Output In Clusters
  cmdump: : Dump confusion matrix in HTML or text formats
  …
```

到此,Mahout 就安装完毕,如果想了解各个算法的入口信息及相应参数,可以查看 MAHOUT_HOME/conf/driver.classes.default.props 文件和 mahout <方法名> -h 命令查看参数。

9.2.3 Mahout 测试

Mahout 安装完成之后,可以通过 Mahout 自带的算法来验证是否能够正常运行,这里使用 Mahout 自带的 *K*-means 算法,具体的 Mahout 测试过程如下。

1. 准备测试数据

本实例的测试数据来自加州大学公开的数据资源集,这里下载了用于聚类测试的数据集 synthetic_control.data 文件(http://archive.ics.uci.edu/ml/datasets.html)。

2. 创建测试目录

下载的数据文件需要复制到 Hadoop 集群的 HDFS 文件系统中,因此要先启动 Hadoop 集群,在 HDFS 上创建测试目录(这里目录名为 testdata),再把数据文件 synthetic_control.data 导入到 testdata 目录中,具体的操作命令如下。

```
$hadoop fs -mkdir testdata                        --创建测试目录 testdata
$hadoop fs -put synthetic_control.data testdata
                                                  --上传测试数据到 HDFS
$mahout org.apache.mahout.clustering.syntheticcontrol.kmeans.Job
                                                  --使用 Mahout 中 K-means 聚类算法
```

```
Running on hadoop, using /opt/hadoop-2.6.0/bin/hadoop and HADOOP_CONF_DIR=
MAHOUT-JOB: /opt/mahout-distribution-0.10.0/mahout-examples-0.10.0-job.jar
...
$hadoop fs -ls output                                  --查看结果
Warning: $HADOOP_HOME is deprecated.

Found 15 items
-rw-r--r--   1 hadoop hadoop      194 2015-07-15 15:24 /output/_policy
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:24 /output/clusteredPoints
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:22 /output/clusters-0
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:22 /output/clusters-1
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:24 /output/clusters-10-final
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-2
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-3
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-4
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-5
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-6
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-7
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-8
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:23 /output/clusters-9
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:22 /hadoop/output/data
drwxrwxrwx   - hadoop hadoop        0 2015-07-15 15:22 /output/random-seeds
```

如果可以看到如上所示的结果，说明 Mahout 已正常运行。上述结果中，clusteredPoints 文件中存放的是最后的聚类的结果，将 cluster-id 和 documents-id 都展示出来了；clusters-N 文件表示第 N 次聚类的结果，其中 N 为某类的样本数目（0～9），clusters-N 结果类型是（Text，Cluster）；data 文件存放的是原始数据。感兴趣的读者可以再尝试其他算法，如 canopy 算法、dirichlet 算法、meanshift 算法等。

9.3　Mahout 算法集

目前，Mahout 已是 Apache Software Foundation（ASF）旗下的顶级开源项目，提供了一些可扩展的机器学习领域经典算法的实现，是比较完备的算法库。Mahout 算法集目前还在不断扩充中，任何对该项目感兴趣的个人或组织都可以加入到该项目的社区中做出贡献。在 Mahout 中实现的机器学习算法如表 9.2 所示。

表 9.2　Mahout 中实现的机器学习算法

算法分类	算法名	中文名
分类算法	Logistic Regression	逻辑回归
	Bayesian	贝叶斯
	Support Vector Machines(SVM)	支持向量机

续表

算法分类	算　法　名	中　文　名
分类算法	Perceptron	感知器算法
	Neural Network	神经网络
	Random Forests	随机森林
	Restricted Boltzmann Machines	有限玻耳兹曼机
	Online Passive Aggressive	在线学习算法
	Hidden Markov Models	隐马尔科夫模型
聚类算法	Canopy Clustering	Canopy 聚类
	K-means Clustering	*K* 均值算法
	Fuzzy *K*-means	模糊 *K* 均值
	Expectation Maximization	EM 聚类(期望最大化聚类)
	Mean Shift Clustering	均值漂移聚类
	Hierarchical Clustering	层次聚类
	Dirichlet Process Clustering	狄里克雷过程聚类
	Latent Dirichlet Allocation	LDA 聚类
	Spectral Clustering	谱聚类
模式挖掘算法	Parallel FP Growth Algorithm	并行 FP Growth 算法
回归	Locally Weighted Linear Regression	局部加权线性回归
降维	Singular Value Decomposition	奇异值分解
	Principal Components Analysis	主成分分析
	Independent Component Analysis	独立成分分析
	Gaussian Discriminative Analysis	高斯判别分析
进化算法	并行化了 Watchmaker 框架	Mahout 0.7 版本中被移除
推荐/协同过滤	Non-distributed recommenders	Taste(UserCF，ItemCF，SlopeOne)
	Collaborative filtering using a parallel matrix factorization	
	Distributed Recommenders	ItemCF
向量相似度计算	RowSimilarityJob	计算列间相似度
	VectorDistanceJob	计算向量间距离
集合方法扩展	Collections	扩展了 Java 的 Collections 类

表 9.2 所列举的算法并不是都并行化处理了，有些还只能运行于单机上，没有被转化为 MapReduce 模式，如 Hidden Markov Models 算法。另外，Mahout 所提供的这些算法

可通过命令进行操作，并不是 Mahout 的核心价值，需要理解算法以及学会怎样运用算法去解决实际场景才是其核心价值。如要了解 Mahout 库中算法的实现机制和并行化原理，可以下载 Mahout 源码版查看相应的算法实现。在查看 Mahout 源码版时应注意，Mahout 项目由多个子项目组成，各子项目分别位于源码的不同目录下，如 mahout-core 是 Mahout 核心模块，位于/core 目录下；mahout-math 用于提供一些数据通用的计算模块，位于/math 目录下；mahout-utils 用于提供一些通用的工具性模块，位于/utils 目录下；examples 是对 Mahout 中各种机器算法的实例。

9.4　分类算法

分类算法是解决分类问题的方法，是数据挖掘、机器学习和模式识别中一个重要的研究领域。分类算法需要先通过对已知类别的分类数据进行训练，从中发现分类规则，以此来预测新数据的类别，从而完成分类。Mahout 中已有多种常用的分类算法，如逻辑回归、贝叶斯、支持向量机、感知器算法、神经网络、随机森林、有限玻耳兹曼机等，其中大多数分类算法都是为了在 Hadoop 集群上运行而编写的。可以使用这些分类算法实现文本分类、客户类别分类、风险评估等应用。下面将对 Mahout 自带的几种分类算法做简单介绍。

9.4.1　逻辑回归

逻辑回归是比较常用的机器学习算法之一，是在线性回归的基础上，套用了一个逻辑函数，利用该函数对未知参数进行估计。在 Mahout 中逻辑回归主要使用随机梯度下降(Stochastic Gradient Decent，SGD)的思想来实现该算法，Mahout 中逻辑回归具体的实现类是 org. apache. mahout. classifier. sgd. TrainLogistic 和 org. apache. mahout. classifier. sgd. RunLogistic，其中，TrainLogistic 是建立模型，RunLogistic 是进行模型评估。下面通过具体的实例来说明 Mahout 中逻辑回归算法如何应用。

首先准备测试数据，本实例的测试数据 donut. csv 使用 Mahout 官网 Logistic Regression 自带的 donut. csv 进行训练，也可从本书配套资料中的 Demo/LogisticRegression 中查看。测试数据准备好之后，上传到 HDFS 上，并通过 trainlogistic 训练模型，在终端运行以下命令。

```
#上传到 HDFS
$ hadoop fs -put donut.csv

#训练模型
$ mahout trainlogistic --input donut.csv \
--output ./model \
--target color --categories 2 \
--predictors x y --types numeric \
--features 20 --passes 100 --rate 50
```

有关 trainlogistic 的参数解释，可在终端输入以下命令查看。

```
$mahout trainlogistic -h
Running on hadoop, using /opt/hadoop-2.6.0/bin/hadoop and HADOOP_CONF_DIR=
MAHOUT-JOB: /opt/mahout-distribution-0.10.0/mahout-examples-0.10.0-job.jar
 Unexpected -h while processing
--help|--quiet|--input|--output|--target|--categories|--predictors|
--types|--passes|--lambda|--rate|--noBias|--features
 Usage:
   [--help --quiet --input <input> --output <output> --target <target>
   --categories <number> --predictors <p1> [<p2> ...] --types
 <t1> [<t2> ...] --passes <passes> --lambda <lambda> --rate <learningRate>
 --noBias --features <numFeatures> ]

--help|--quiet|--input|--output|--target|--categories|--predictors|
--types|--passes|--lambda|--rate|--noBias|--features
     ...(中间部分省略)
15/07/17 16:26:48 INFO MahoutDriver: Program took 88 ms (Minutes:
0.0014666666666666667)
```

其中，input 为输入数据；output 为输出模型文件；--target 预测的变量（输入数据要求第一行为变量名）；categories 为预测变量的取值个数；predictors 为参与建模的变量；types 为预测变量的类型（numeric、word、text 类型中的其中一个类型）；passes 为训练时对输入数据测试的次数；features 为内部随机向量维度；rate 为学习速率。

然后使用模型评估 RunLogistic 命令来预测结果，其中测试数据可使用 Mahout 官网 LogisticRegression 自带的 donut-test. csv 进行评估，也可从本书配套资料中的 Demo/LogisticRegression 中查看，具体的评估命令如下。

```
$mahout runlogistic --input donut-test.csv \
--model ./model \
--scores \
--auc \
--confusion
```

其中，input 就是测试数据；model 是模型文件；scores 打印预测值和原始值对比结果；auc 打印 auc 值；confusion 打印模糊矩阵。有关逻辑回归算法的基本原理及本实例的运行结果请读者查看有关机器学习的书籍，并自己动手实践加以理解。

9.4.2 贝叶斯

贝叶斯分类是一种十分简单的算法，源于古典概率理论，是通过某对象的先验概率，利用贝叶斯公式计算出其后验概率，即该对象属于某一类的概率，选择具有最大后验概率的类作为该对象所属的类。这类算法均以贝叶斯定理为基础，故统称为贝叶斯分类。贝



叶斯定理如下：

$$p(B \mid A)=\frac{p(A \mid B)p(B)}{p(A)}$$

贝叶斯做了一个简单的前提假设，即给定目标值属性之间相互条件独立（属性间不存在依赖关系）。其中，$p(A|B)$表示事件 B 发生的前提下，事件 A 发生的概率；$p(A)$表示事件 A 发生的概率；$p(B)$表示事件 B 发生的概率。这样就可以求得事件 A 发生的前提下，事件 B 发生的概率。贝叶斯定理给出了最小化误差的最优解决方法，可用于分类和预测。

Mahout 主要实现了两种贝叶斯分类器，即朴素贝叶斯算法（Traditional Naive Bayes）和互补型朴素贝叶斯算法（Complementary Naive Bayes）。后一种贝叶斯分类器是在朴素贝叶斯基础上增加了结果分析功能（org. apache. mahout. classifier. ResultAnalyzer 类）。在 Mahout 中，与贝叶斯分类器有关的主要类有 BayesUtils、NaiveBayesModel、StandardNaiveBayesClassifier、ComplementaryNaiveBayesClassifier（相应的类在 org. apache. mahout. classifier. naivebayes 包中）。有兴趣的读者可以查看相应源码，查看其基本原理和并行化实现方式。下面通过具体的实例来说明 Mahout 中贝叶斯算法如何应用。

本实例的测试数据由 20 个新闻组数据组成，收集大约两万个新闻组文档，均匀地分布在 20 个不同的集合中。我们将使用 Mahout 的 Bayes Classifier 创造一个模型，实现将一个新文档分类到这 20 个新闻组集合中。数据集 20news-bydate. tar. gz 文件可从本书配套资料中的 Demo/NativeBayes 中查看，数据准备好之后，首先要对数据进行解压，具体的操作命令如下。

```
#解压数据文件
$ tar -zxvf 20news-bydate.tar.gz

#上传到 HDFS
$ hadoop fs -put 20news-bydate-test
$ hadoop fs -put 20news-bydate-train
```

解压后的数据集按时间分为训练数据（20news-bydate-train）和测试数据（20news-bydate-test），每个数据文件为一条信息，文件头部几行指定消息的发送者、长度、类型、使用软件以及主题等，然后用空行将其与正文隔开，正文没有固定格式。

下一步需要将数据集进行转换，因为使用 Mahout 自带的示例程序对朴素贝叶斯模型进行训练的类 TrainNaiveBayesJob 的输入是经过 mahout seqdirectory 和 mahout seq2sparse 向量化的序列化文件，输出是一个 model（训练器）。seqdirectory 对应的源文件是 org. apache. mahout. text. SequenceFilesFromDirectory，使用 mahout seqdirectory 命令可以将文本文件转成 Hadoop 的 SequenceFile 文件，SequenceFile 文件是一种二进制存储的 key-value 键值对；seq2sparse 是将上面生成的 SequenceFile 转成向量文件，对应的源文件是 org. apache. mahout. vectorizer. SparseVectorsFromSequenceFiles。seq2sparse 实际上是一个计算文本 TF，DF，TFIDF 的过程，依次产生的向量文件目录结

构如下。

(1) tokenized-documents 目录：保存着分词过后的文本信息。

(2) wordcount 目录：保存着全局的词汇出现的次数。

(3) dictionary. file-0 目录：保存着这些文本的词汇表。

(4) tf-vectors 目录：保存着以 TF 作为权值的文本向量。

(5) df-count 目录：保存着文本的频率信息。

(6) frequcency-file-0 目录：保存着词汇表对应的频率信息。

(7) tfidf-vectors 目录：保存着以 TFIDF 作为权值的文本向量。

将数据集转换为向量文件的具体操作命令如下。

```
#转换为序列文件(sequence files)
$mahout seqdirectory -i 20news-bydate-train -o 20news-bydate-train-seq
$mahout seqdirectory -i 20news-bydate-test -o 20news-bydate-test-seq

#转换为 tf-idf 向量
$mahout seq2sparse -i 20news-bydate-train-seq \
 -o 20news-bydate-train-vector -lnorm -nv -wt tfidf
$mahout seq2sparse -i 20news-bydate-test-seq \
 -o 20news-bydate-test-vector -lnorm -nv -wt tfidf
```

最后，训练和检验朴素贝叶斯模型，并查看训练后的结构，具体的操作命令如下。

```
#训练朴素贝叶斯模型
$mahout trainnb -i 20news-bydate-train-vectors/tfidf-vectors \
-el -o model -li labelindex -ow

#测试朴素贝叶斯模型
$mahout testnb -i 20news-bydate-train-vectors/tfidf-vectors \
-m model -l labelindex -ow -o test-result

#查看训练后的结构
$mahout seqdumper -i labelindex
Input Path: labelindex
Key class: class org.apache.hadoop.io.Text Value Class: class org.apache.hadoop.io.IntWritable
Key: alt.atheism: Value: 0
Key: comp.graphics: Value: 1
Key: comp.os.ms-windows.misc: Value: 2
Key: comp.sys.ibm.pc.hardware: Value: 3
Key: comp.sys.mac.hardware: Value: 4
Key: comp.windows.x: Value: 5
Key: misc.forsale: Value: 6
```




```
Key: rec.autos: Value: 7
Key: rec.motorcycles: Value: 8
Key: rec.sport.baseball: Value: 9
Key: rec.sport.hockey: Value: 10
Key: sci.crypt: Value: 11
Key: sci.electronics: Value: 12
Key: sci.med: Value: 13
Key: sci.space: Value: 14
Key: soc.religion.christian: Value: 15
Key: talk.politics.guns: Value: 16
Key: talk.politics.mideast: Value: 17
Key: talk.politics.misc: Value: 18
Key: talk.religion.misc: Value: 19
Count: 20
```

该算法可用来处理大规模数据的分类，对于特征值选取越准确的，正确率会越高，算法简单容易理解。有关贝叶斯算法的基本原理及本实例的运行结果，请读者查看有关机器学习的书籍并自己动手实践加以理解。

9.4.3 随机森林

随机森林算法是 Leo Breiman 于 2001 年提出的一种新型分类和预测模型，是用随机的方式建立一个森林，森林里有很多决策树，每个决策树之间没有关联，当有一个新的输入样本进入时，就让森林中的每棵决策树分别进行判断该样本应该属于哪一类，被选择最多的一类就是该样本所属的类。Mahout 实现了两种随机森林算法，一种是 MapReduce 模式的(Partial，请查看 org. apache. mahout. classifier. df. mapreduce. partial 包中的相关类)，另一种是单机运行模式的(Breiman，请查看 org. apache. mahout. classifier. df. BreimanExample 类)。这里以 MapReduce 模式的随机森林算法 Partial 为例，介绍 Mahout 中随机森林的用法。

Mahout 中 MapReduce 模式随机森林 Partial 的实现主要分为以下三步。

第一步，对数据进行描述(org. apache. mahout. classifier. df. tools. Describe)，即输入训练集，对数据进行描述后生成. info 文件。

第二步，生成决策森林(org. apache. mahout. classifier. df. mapreduce. BuildForest)，输入为训练集和第一步生成的. info 文件，运行后生成 forest. seq 文件，该文件记录了生成的所有决策树。

第三步，预测结果(org. apache. mahout. classifier. df. mapreduce. TestForest)，输入为测试集和决策森林(forest. seq)，运行后生成测试集的预测结果及统计预测结果的准确率。下面通过具体的实例来说明 Mahout 中随机森林算法如何应用。

本实例的数据集 KDDTrain＋. arff 和 KDDTest＋. arff 文件可从本书配套资料中的 Demo/RandomForests 中查看，数据准备好之后，打开数据文件，删除其中以@开头的数据，并将数据上传到 HDFS，具体的操作命令如下。

```
#上传到 HDFS
$ hadoop fs -put KDDTrain+ .arff
$ hadoop fs -put KDDTest+ .arff
```

其中,KDDTrain+.arff 为训练数据,KDDTest+.arff 为测试数据。然后执行随机森林算法的描述(Describe)、训练(BuildForest)和预测(TestForest)三个步骤。描述(Describe)的具体操作命令如下。

```
#生成数据集描述文件
$ mahout org.apache.mahout.classifier.df.tools.Describe -p KDDTrain+ .arff \
-f /testdata/KDDTrain+ .info -d N 3 C 2 N C 4 N C 8 N 2 C 19 N L
```

在 Describe 中,-p 为训练集路径;-f 为输出描述文件的路径;-d 为训练集中每条记录属性的类型串。最后的“N 3 C 2 N C 4 N C 8 N 2 C I 9 N L”字符用来描述数据属性,如 N 是 Numerical 的缩写;L 是 Label 的缩写;C 是 Categorical 的缩写;I 是 Ignore 的缩写;9 表示 9 个都是 N。

当生成数据集描述文件之后,下一步将要对数据集进行训练,即生成决策森林,具体的操作命令如下。

```
#生成决策森林
$ mahout org.apache.mahout.classifier.df.mapreduce.BuildForest \
 -Dmapred.max.split.size=1874231 \
 -d KDDTrain+ .arff -ds KDDTrain+ .info -sl 5 -p -t 100 -o /nsl-forest
```

其中,-sl 表示属性数目;-t 表示构建多少棵树;-d 表示训练数据文件;-ds 表示数据集地址;-o 表示模型输出地址;-p 表示平行计算,默认用内存模式,如果数据量比较大,则启动该配置。

最后,对测试数据进行分类,即使用决策森林分类新数据,具体的操作命令如下。

```
#使用决策森林分类新数据
$ mahout org.apache.mahout.classifier.df.mapreduce.TestForest \
-i KDDTest+ .arff -ds KDDTrain+ .info -m /nsl-forest -a -mr -o /predictions
```

其中,-i 表示测试文件路径;-ds 表示数据描述文件路径;-m 表示决策森林文件所在路径;-mr 表示使用 MapReduce 进行分布式计算;-o 表示输出文件路径。本实例的运行结果请读者自己动手实践加以理解。有关更多的 Mahout 分类算法,有兴趣的读者可以认真查看 org.apache.mahout.classifier 包中的相应类。

9.5 聚类算法

聚类是将数据集划分为若干相似对象组成的多个组或簇的过程,使得同一组中对象间的相似度最大化,不同组中对象间的相似度最小化,即组内同质,组间差异。数据对象



间相似性的衡量有很多经典算法，如欧几里得距离算法、余弦距离算法、皮尔逊相关系数算法等（Mahout 提供了几种常见的距离度量实现，请查看 org. apache. mahout. common. distance 包下的相应类）。Mahout 中已基于这些相似度衡量实现了多种常用的聚类算法，如 Canopy 聚类、K 均值、层次聚类等，可以使用这些聚类算法实现文本聚类、客户类别聚类等应用场景中。下面将对 Mahout 自带的几种聚类算法做简单介绍。

9.5.1　Canopy 聚类

Canopy 聚类算法是将每个对象用多维特征空间里的一个点来表示，通过一个快速近似距离度量和两个距离阈值 $T_1>T_2$ 将对象分组到类的一种简单快捷的聚类算法。该算法的基本思想是设置一个空的 Canopy 集合，将第一个点作为集合中的一个点，接着读取下一个点与该集合中的每个点计算距离 dist，若 dist$<T_1$，则为 Canopy 集合中的一点；若 dist$>T_1$，则不能作为 Canopy 集合中的一点；若 dist$<T_2$，则标记该点已于该 Canopy 集合强关联；若该点不存在强关联的 Canopy 集合中的点，则为该点创建一个新的 Canopy 集合，直到所有点都遍历完为止。从 Canopy 算法的基本思想可以看出，dist$<T_2$ 点属于有且仅有一个簇，$T_2<$dist$<T_1$ 的点可能属于多个簇。

Mahout 中 Canopy 算法的实现主要分为以下三步。

第一步，将输入的数据处理为 Canopy 算法可使用的输入格式（Mahout 的聚类分类过程中，需要将数据转化成向量 Vector，其实现的接口是 org. apache. mahout. math. Vector，也可通过继承 Mahout 中 AbstractVector 来实现自己的向量模型）。

第二步，将每个 Mapper 对划分到自己的点根据阈值 T_1 和 T_2 来标记 Canopy，输出每个 Canopy 的中心向量（该过程由 org. apache. mahout. clustering. canopy. CanopyMapper 类实现），每个 Reducer 再接收来自 Mapper 的中心向量，加以整合并计算出最后的 Canopy 的中心向量（该过程由 org. apache. mahout. clustering. canopy. CanopyReducer 类实现）。

第三步，使用 Reducer 计算出中心向量，采用最近距离原则对原始数据进行聚类（该过程由 org. apache. mahout. clustering. canopy. ClusterDriver 类实现）。下面通过具体的实例来说明 Mahout 中 Canopy 算法如何应用。

本实例的数据集来自加州大学公开的数据资源集，这里下载了用于聚类测试的数据集 reuters21578. tar. gz 文件（http://kdd. ics. uci. edu/databases/reuters21578/reuters21578. tar. gz），也可从本书配套资料中的 Demo/Canopy 文件夹中查看。该数据集包含路透社共 21 578 篇新闻报告，通过专业人员手工标注的形式化语料库存。首先要将该文件进行解压缩，具体的操作命令如下。

```
#解压缩数据
$ tar -zxvf reuters21578.tar.gz
```

该数据文件解压后会出现“*.sgm”格式的文件，这种格式的文件本质上是 XML 格式的文件，需要将这些文件转化为 SequenceFile 格式的文件，可使用 Mahout 自带的工具

类实现文本抽取(org. apache. lucene. benchmark. utils. ExtractReuters 类),会对文本内容进行提取(抽取标题<TITLE>和正文<BODY>中的文本),提取后的文本保存在 reuters-out 目录下。然后将 reuters-sgm 和 reuters-out 上传到 HDFS,具体的操作命令如下。

```
#抽取文本内容
$mahout org.apache.lucene.benchmark.utils.ExtractReuters
./reuters-sgm
./reuters-out

#上传到 HDFS
$hadoop fs -put reuters-sgm
$hadoop fs -put reuters-out
```

下一步需要将数据集进行转换,因为 Mahout 聚类算法的输入为 List<Vector>,即需要将每个待聚类的文档表示为向量形式,转换后输出的文件格式为<Text,Text>,具体的操作命令如下。

```
#转化为 SequenceFile 文件
$MAHOUT seqdirectory \
 -i ${WORK_DIR}/reuters-out \
 -o ${WORK_DIR}/reuters-out-seqdir \
 -c UTF-8 \
 -chunk 64 \
 -xm sequential

#转换为向量表示
$MAHOUT seq2sparse \
 -i ${WORK_DIR}/reuters-out-seqdir/
 -o ${WORK_DIR}/reuters-out-seqdir-sparse-canopy \
 -ow \
 --weight tfidf \
 --maxDFPercent 85 \
 --namedVector
```

使用 Mahout 的 seqdirectory 命令将原始语料转换成 SequenceFile 文件格式,charset 为 UTF-8;chunkSize 为 64MB;xm 选择执行的方法为 sequential(表示在本地顺序执行,也可以使用 MapReduce)。使用 seq2sparse 将 SequenceFile 文件转化为向量表示,-i 和-o 表示输入和输出;-ow(-overwrite)表示即使输出目录存在,也进行覆盖操作;--weight (--wt)表示权重公式;--maxDFPercent 85 表示过滤高频词,当 DF 大于 85%时,将不再作为词特征输出到向量中;--namedVector 表示向量会输出附加信息。最后,使用 Mahout 的 Canopy 算法对测试数据进行聚类,具体的操作命令如下。

```
#Canopy 聚类
$ mahout canopy \
  - i $ {WORK_DIR}/tfidf- vectors \
  - o $ {WORK_DIR}/reuters- canopy- centroids \
  - dm org.apache.mahout.common.distance.CosineDistanceMeasure \
  - t1 150 \
  - t2 80 \
  - ow \
  - - clustering
```

其中，输入文件使用的是转换后的序列文件；距离计算方式使用的是欧式距离；T_1 和 T_2 分别设置为 150 和 80；--clustering 选项表示最后对原始数据进行分类。本实例的运行结果可在 reuters-canopy-centroids 目录中查看，请读者自己动手实践加以理解。

9.5.2　K-means 聚类

K-means 聚类是数据挖掘最为经典的基于划分的聚类算法，该算法的基本思想是以空间中 *K* 个点为中心进行聚类，把所剩下的其他点与这 *K* 个点进行相似度(距离)计算，分别将它们分配给与其最相似的聚类，然后再计算每个所获取新聚类的聚类中心，不断重复该过程直到标准测度函数(一般使用误差平方和准则函数)开始收敛为止。

在 Mahout 中的 *K*-means 算法其实就是将 *K*-means 的算法通过 MapReduce 并行化，其中 Map 用来读取分配到该点的每条数据，与中心做对比，求出该点对应的中心，然后以中心 ID 为 Key，该点为 Value 将数据输出；Reduce 是将相同的 Key 归并到一起，集中与该 Key 对应的点，再求出这些点的平均值，输出新的聚类中心。Mahout 中 *K*-means 聚类过程与其他聚类过程相同，也要经过数据集转化为 SequenceFile→seq2sparse→执行算法→查看结果的过程，这里就不再赘述。有关 *K*-means 的聚类实例，可查看 Mahout 安装目录的{home}/examples/bin/cluster-reuters. sh 文件，该文件是关于各种 *K*-means 聚类的具体实例，如 *K*-means、fuzzykmeans、LDA (Latent Dirichlet Allocation)、streamingkmeans。该文件 cluster-reuters. sh 的执行具体流程请查看源码，这里使用文本编辑器打开 cluster-reuters. sh 文件，并给出部分重要源码及注释，方便读者理解。

```
#自动下载路透社新闻语料并进行解压
echo "Downloading Reuters- 21578"
curl http://kdd.ics.uci.edu/databases/reuters21578/reuters21578.tar.gz
  - o $ {WORK_DIR}/reuters21578.tar.gz

#调用 Mahout 自带的 ExtractorReuters 进行内容抽取
$ MAHOUT org.apache.lucene.benchmark.utils.ExtractReuters $ {WORK_DIR}/reuters- sgm $ {WORK_DIR}/
reuters- out
```

```
#调用 seqdirectory,将其转换成 SequenceFile 文件
$MAHOUT seqdirectory -i ${WORK_DIR}/reuters-out -o ${WORK_DIR}/reuters-out-seqdir -c UTF-8 -
chunk 64 -xm sequential

#调用 seq2sparse,将 SequenceFile 文件转换为向量文件
$MAHOUT seq2sparse \
-i ${WORK_DIR}/reuters-out-seqdir/ \
-o ${WORK_DIR}/reuters-out-seqdir-sparse-kmeans --maxDFPercent 85
-- namedVector \

#调用 K-means 算法
$MAHOUT kmeans \
    -i ${WORK_DIR}/reuters-out-seqdir-sparse-kmeans/tfidf-vectors/ \
    -c ${WORK_DIR}/reuters-kmeans-clusters \
    -o ${WORK_DIR}/reuters-kmeans \
    -dm org.apache.mahout.common.distance.EuclideanDistanceMeasure \
    -x 10 -k 20 -ow --clustering \
$MAHOUT clusterdump \                            #调用 clusterdump 分析聚类结果
    -i '$DFS -ls -d ${WORK_DIR}/reuters-kmeans/clusters-*-final | awk '{print $8}'' \
    -o ${WORK_DIR}/reuters-kmeans/clusterdump \
    -d ${WORK_DIR}/reuters-out-seqdir-sparse-kmeans/dictionary.file-0 \
    -dt sequencefile -b 100 -n 20 --evaluate -dm org.apache.mahout.common.distance.
    EuclideanDistanceMeasure -sp 0 \
--pointsDir ${WORK_DIR}/reuters-kmeans/clusteredPoints \

#调用 fuzzykmeans 聚类算法
$MAHOUT fkmeans \
    -i ${WORK_DIR}/reuters-out-seqdir-sparse-fkmeans/tfidf-vectors/ \
    -c ${WORK_DIR}/reuters-fkmeans-clusters \
    -o ${WORK_DIR}/reuters-fkmeans \
    -dm org.apache.mahout.common.distance.EuclideanDistanceMeasure \
    -x 10 -k 20 -ow -m 1.1 \
$MAHOUT clusterdump \                            #调用 clusterdump 分析聚类结果
    -i ${WORK_DIR}/reuters-fkmeans/clusters-*-final \
    -o ${WORK_DIR}/reuters-fkmeans/clusterdump \
    -d ${WORK_DIR}/reuters-out-seqdir-sparse-fkmeans/dictionary.file-0 \
    -dt sequencefile -b 100 -n 20 -sp 0 \

#调用 LDA 聚类算法
$MAHOUT cvb \
    -i ${WORK_DIR}/reuters-out-matrix/matrix \
    -o ${WORK_DIR}/reuters-lda -k 20 -ow -x 20 \
    -dict ${WORK_DIR}/reuters-out-seqdir-sparse-lda/dictionary.file-* \
```




```
        -dt ${WORK_DIR}/reuters-lda-topics \
        -mt ${WORK_DIR}/reuters-lda-model \
    $MAHOUT vectordump \
        -i ${WORK_DIR}/reuters-lda-topics/part-m-00000 \
        -o ${WORK_DIR}/reuters-lda/vectordump \
        -vs 10 -p true \
        -d ${WORK_DIR}/reuters-out-seqdir-sparse-lda/dictionary.file-*  \
        -dt sequencefile -sort ${WORK_DIR}/reuters-lda-topics/part-m-00000 \

    #调用 streamingkmeans 聚类算法
    $MAHOUT streamingkmeans \
        -i ${WORK_DIR}/reuters-out-seqdir-sparse-streamingkmeans/tfidf- vectors/ \
        --tempDir ${WORK_DIR}/tmp \
        -o ${WORK_DIR}/reuters-streamingkmeans \
        -sc org.apache.mahout.math.neighborhood.FastProjectionSearch \
        -dm org.apache.mahout.common.distance.SquaredEuclideanDistanceMeasure \
        -k 10 -km 100 -ow \
    $MAHOUT qualcluster \
        -i ${WORK_DIR}/reuters-out-seqdir-sparse-streamingkmeans/tfidf- vectors/part-r-00000 \
        -c ${WORK_DIR}/reuters-streamingkmeans/part-r-00000    \
        -o ${WORK_DIR}/reuters-cluster-distance.csv \
        && \
```

除了上面介绍的几种常用的聚类算法之外，还有很多其他聚类算法，如 EM 聚类(Expectation Maximization，期望最大化聚类)、均值漂移聚类(Mean Shift Clustering)、层次聚类(Hierarchical Clustering)、狄里克雷过程聚类(Dirichlet Process Clustering)、谱聚类(Spectral Clustering)等。有关 Mahout 中更多聚类算法，可以认真查看 org. apache. mahout. clustering 包中的相应类。

9.6　模式挖掘算法

模式挖掘算法是一种常被用来进行关联分析和挖掘频繁项的算法。该算法使用了一种称为频繁模式树(Frequent Pattern Tree)的数据结构，其数据结构为＜频繁项头，项前缀树＞。前缀树是一种存储候选项集的数据结构，树的分支用项名标识，树的节点存储后缀项，项集由路径组成。该算法需要扫描两次数据库，第一次扫描数据库计算所有项目的频数并进行降序排列生成一个 F-List，第二次扫描将数据压缩成一个频繁模式树 FP-Tree，接着通过 FP-Growth 来递归挖掘该 FP-Tree，从而寻找出频繁项目集。

在 Mahout 中的模式挖掘算法(Parallel FP Growth Algorithm)是一种并行化处理的模式挖掘算法，解决了单一节点的 FP-Growth 算法在时间和空间上的双重瓶颈。在 Mahout 0.8 之后的版本已经将该算法移除，如需要更多地了解该算法请查看 Mahout 0.8 以下版本中的 FPGrowthDriver、PFPGrowth、ParallelCountingMapper、

TransactionTree、ParallelCountingReducer、ParallelFPGrowthMapper、ParallelFPGrowthReducer 等类，并结合 Haoyuan Li 的论文 *Parallel FP-Growth for Query Recommendation*[87]来理解如何使用 MapReduce 并行化 FP-Growth 算法。有关模式挖掘算法的实例数据集，可从本书配套资料中的 Demo/Parallel FP-Growth 文件夹中查看（retail.dat 为测试数据集，retail_results_with_min_sup_100.dat 为结果集）。首先在 HDFS 中新建 input 目录，然后将 retail.dat 数据集上传到 input 目录中，再使用 Mahout 的模式挖掘算法对测试数据进行模式挖掘，具体的操作命令如下。

```
$mahout fpg -i input/retail.dat \
 -o output \
 -method mapreduce \
 -regex '[\ ]' -s 2
```

其中，-i 表示输入路径；-o 表示输出路径；-method 表示计算方法，即单机模式还是分布式；-regex 表示正则表达式；-minSupport 表示最小支持度阈值，默认值为 3。执行完上述命令后，会在 output 目录下生成 4 个文件夹 fList，frequentpatterns，fpGrowth，parallelcounting，查看执行结果的具体操作命令如下。

```
$hadoop fs -ls -R output                           #查看 output 目录下生成的文件夹
$mahout seqdumper -i output/frequentpatterns/part-r-00000
                                                   #查看执行结果
```

Mahout 中的并行化 FP-Growth 算法解决了面对大量数据时传统 FP-Growth 的时间和空间的性能瓶颈，有兴趣的读者可以考虑在并行化 FP-Growth 算法的基础上，如何更好地考虑负载均衡以及优化频繁模式的表示等问题。

9.7 协同过滤算法

协同过滤算法（Collaborative Filtering）也称为推荐算法，是根据目标用户的行为特征，为其发现一个兴趣相投、属性相似的群体，然后根据群体的喜好来为目标用户过滤可能感兴趣的内容。Mahout 完整地封装了并行化的协同过滤算法，并提供了 API，供开发人员根据自己的业务场景进行算法配置和调优。要基于 Mahout 实现协同过滤，就需要经过收集用户偏好→计算用户/物品相似度→计算推荐物品等过程。下面将对每个过程进行简单介绍。

9.7.1 收集用户偏好

要从用户的行为和偏好中发现规律，并基于此给予推荐，首先就需要收集用户的偏好信息并进行有效的存储和表达，该过程是协同过滤算法推荐效果最基础的决定因素。用户可以有很多方式来提供自己的偏好信息，而且不同的应用也可能大不相同。表 9.3 给出了一种用户行为和用户偏好的表达方式。

表 9.3　用户行为和偏好

用户行为	类型	特　征	作　用
评分	显式	整数量化的偏好，可取值[0,10]	通过用户对物品的评分，可以精确地得到用户的偏好
投票	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的投票，可以较精确地得到用户的偏好
转发	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的转发，可以较精确地得到用户的偏好
收藏	显式	布尔量化的偏好，取值是 0 或 1	通过用户对物品的收藏，可以精确地得到用户的偏好
评论	显式	一段文字，需要进行文本分析得到偏好	通过分析用户的评论，可以得到用户的情感：喜欢还是讨厌

表 9.3 的实例是较为通用的一种表达方式，具体的设计可以根据应用场景的特点添加特殊的用户行为，并表示用户对物品的偏好程度。当收集到用户行为数据之后，还要对这些数据进行预处理，如减噪和归一化处理，再根据不同应用的行为分析进行选择分组或者加权处理，得到一个用户偏好的二维矩阵＜用户列表，物品列表＞，其值是用户对物品的偏好。在 Mahout 中是通过偏好（Perference）和数据模型（Data Model）这两种数据表示形式实现上述过程的。

1. 偏好

在 Mahout 中用户的偏好被抽象为一个 Perference 接口，该接口包含 userID、itemId 和偏好值，Perference 接口的通用实现为 GenericPreference 类。Mahout 还分装了一个 PerferenceArray 接口，用于保存一组用户偏好数据，其实现类为 GenericUserPreferenceArray 类和 GenericItemPreferenceArray 类。有兴趣的读者可以查看 org.apache.mahout.cf.taste.impl.model 包中的相应类。

2. 数据模型

Mahout 协同过滤接受的数据输入类型为 DataModel，实现从任意类型的数据源抽取用户偏好信息，并返回从输入的偏好数据中关联到一个物品用户 ID 列表和 count 计数，以及输入数据中所有用户和物品的数量。DataModel 的实体实现类包括支持文件读取的 FileDataModel 类、内存版的 GenericDataModel 类和支持数据库读取的 JDBCDataModel 类。有兴趣的读者可以查看 org.apache.mahout.cf.taste.impl.model 包中的相应类。

9.7.2　相似度计算

不管是基于用户的协同过滤算法 UserCF，还是基于物品的协同过滤算法 ItemCF，首先都要对用户或物品进行相似度计算，从而发现兴趣相似的用户或物品。在 Mahout 中，通常用 Jaccard 系数（也可称为 Tanimoto 系数，Tanimoto Coefficient）或余弦相似度（Cosine Similarity）计算来解决相似度问题，也可用如欧几里得距离（Euclidean

Distance)、皮尔森相关系数(Perarson Correlation Coefficient)等方法解决。不管用哪种方式解决,本质上还是通过计算两个向量之间的距离来衡量两个用户(UserCF)或物品(ItemCF)之间的相似度。下面先了解一下 Mahout 中几种常用的相似度计算方法。

1. 欧几里得距离

欧几里得距离最初用于计算欧几里得空间中两点的距离,假设 x 和 y 是 N 维空间的两个点,它们之间的欧几里得距离为:

$$d(x,y) = \sqrt{\left(\sum (x_i - y_i)^2\right)}$$

当 $N=2$ 时,欧几里得距离就是平面上两点之间的欧式距离,两点之间的距离越小,其相似度越大,可通过如下公式表示其相似度大小。

$$\mathrm{sim}(x,y) = \frac{1}{1+d(x,y)}$$

2. 皮尔森相关系数

皮尔森相关系数也称为皮尔森积矩相关系数,是用来反映两个变量线性相关程度的统计量,它的取值在[−1,+1]之间,其计算公式如下。

$$p(x,y) = \frac{\sum x_i y_i - n\overline{xy}}{(x-1)s_x s_y} = \frac{n\sum x_i y_i - \sum x_i y_i}{\sqrt{n\sum x_i^2 - \left(\sum x_i\right)^2} \times \sqrt{n\sum y_i^2 - \left(\sum y_i\right)^2}}$$

其中,s_x 和 s_y 是 x 和 y 的样品标准偏差。

3. 余弦相似度

余弦相似度是利用向量空间中两个向量夹角的余弦值作为衡量两个个体间差异的大小。余弦相似度更加注重两个向量在方向上的差异,而非距离上或长度上的差异。余弦相似度的计算公式如下。

$$\mathrm{sim}(x,y) = \cos\theta = \frac{\vec{x} \cdot \vec{y}}{\|x\| \cdot \|y\|}$$

如果对两个项的属性进行数据中心化,则余弦相似度和皮尔森相似度是一样的。Mahout 实现了数据中心化过程,因此皮尔森相关系数也是数据中心化后的余弦相似度。在新版本的 Mahout 中,也提供了计算非中心化的余弦相似度类(UncenteredCosineSimilarity 类)。

4. Jaccard 系数

Jaccard 系数也可称为 Tanimoto 系数,主要用于计算符号度量或布尔度量的个体间的相似度,因此该度量只关心个体间共同具有的特征是否一致,而无法衡量差异具体值的大小。Jaccard 系数的计算公式如下。

$$\mathrm{Jaccard}(x,y) = \frac{x \cap y}{x \cup y}$$

Mahout 主要提供了针对用户相似度和物品相似度的计算,如用于计算欧几里得距

离的类 EuclideanDistanceSimilarity，用于计算皮尔森相关系数的类 PearsonCorrelationSimilarity，用于计算余弦相似度类的类 UncenteredCosineSimilarity 类，用于计算 Jaccard 系数（Tanimoto 系数）的类 TanimotoCoefficientSimilarity。这些类都继承自 AbstractSimilarity 类。该类中有三个重要方法：userSimilarity()、itemSimilarity()、computeResult()。其中，userSimilarity()和 itemSimilarity()方法会在计算好相应变量之后调用子类实现 computeResult()方法。如果读者想更深入地了解相似度的计算，请结合上述理论并结合相关代码理解。

9.7.3 推荐计算

经过前面的相似度计算已经可以得到相邻用户和相邻物品，下面就要基于这些信息为目标用户进行推荐。Mahout 提供了多种类型的计算推荐，如基于用户的推荐(GenericUserBasedRecommender)、基于物品的推荐(GenericItemBasedRecommender)、基于 Slope-one 算法的推荐（SlopeOneRecommender）、基于 SVD 算法的推荐(SVDRecommender)、基于最近邻算法的推荐(KnnItemBasedRecommender)、基于聚类的推荐(TreeClusteringRecommender)。这几种计算推荐各有优缺点，实际项目中需要根据不同的应用场景来选择适合的计算推荐引擎。下面介绍 Mahout 中常用的基于用户的推荐 UserCF 和基于物品的推荐 ItemCF。

1. 基于用户的推荐 UserCF

基于用户的协同过滤是找到与目标用户对物品偏好相似的相邻用户集合，然后将用户集合中相邻用户所喜欢的物品且目标用户没有关注的物品推荐给目标用户，其基本思想是将目标用户对所有物品的偏好作为一个向量来计算用户之间的相似度，找到 K 个邻居用户后，根据邻居用户的相似度权重及他们对物品的偏好，预测目标用户没有偏好的未涉及物品，从而计算得到一个排序的物品列表作为推荐。Mahout 中基于用户的推荐 UserCF 的实现类为 GenericUserBasedRecommender 类，该类的输入参数为 DataModel、UserNeighborhood、UserSimilarity。其中，DataModel 是用户偏好信息的抽象接口，支持从任意类型的数据源抽取用户偏好信息；UserSimilarity 用于定义两个用户间的相似度，可用来计算用户的相似邻居；UserNeighborhood 用于基于用户相似度的推荐方法中，推荐的内容是基于找到与当前用户偏好相似的邻居用户的偏好物品。基于用户的推荐 UserCF 的具体过程如下。

(1) 查询与该用户相似的用户，以及相似用户与该用户的相似度。其中，相似用户和相似度是进行推荐计算的根据。

(2) 应用 UserNeighborhood 获取指定用户 $User_i$ 最相似的 K 个邻居用户的集合 $\{User_1, User_2, \cdots, User_k\}$。

(3) 根据 K 个邻居用户的偏好物品集合 $\{Item_0, Item_1, \cdots, Item_m\}$ 计算用户 $User_i$ 对每个 $Item_j$ 的偏好程度 $perf(User_i, Item_j)$，其偏好程度的计算公式如下。

$$\text{perf}(\text{User}_i, \text{Item}_j) = \frac{\sum_{l=1}^{k} \text{sim}(\text{User}_l, \text{User}_i) \times p(\text{User}_l, \text{Item}_j)}{\sum_{l=1}^{k} \text{sim}(\text{User}_l, \text{User}_i)}$$

其中，$p(\text{User}_l, \text{Item}_j)$是 User_l 对 Item_j 的偏好值。

(4) 根据 User_i 对每个 Item_j 的偏好程度 $\text{Perf}(\text{User}_i, \text{Item}_j)$的数值从高到低排序，把前 N 个 Item 推荐给 User_i。

下面通过具体的实例来说明如何实现基于用户的推荐。本实例的数据来源于网站 http://www.grouplens.org/中的 MovieLens 压缩文件，该文件夹下有三个文件：movies.dat、ratings.dat 和 users.dat。其中，movies.dat 的文件描述是：电影编号::电影名::电影类别;ratings.dat 的文件描述是：用户编号::电影编号::电影评分::时间戳;users.dat 的文件描述是：用户编号::性别::年龄::职业::Zip-code。这些文件包含来自 6040 个 MovieLens 用户在 2000 年对约 3900 部电影的 1 000 209 个匿名评分信息。本实例直接使用 ratings.dat 文件作为数据源，基于用户的推荐引擎实现代码如下。

```
public class UserCFRecommender {
    //step:1.构建数据模型 2.计算相似度 3.查找 k紧邻 4.构造推荐引擎
    //构造数据模型
    DataModel model=new FileDataModel(new File("ratings.dat"));

    //用 PearsonCorrelation算法计算用户相似度
    UserSimilarity similarity=new PearsonCorrelationSimilarity(model);

    /*计算用户的"邻居",这里将与该用户最近距离为 3的用户设置为该用户的"邻居"*/
    UserNeighborhood neighborhood=new NearestNUserNeighborhood(3, similarity, model);

    /*构建一个 GenericUserBasedRecommender 推荐器需要数据源 (DataModel)、用户相似性
(UserSimilarity)和相邻用户相似度 (UserNeighborhood) */
    Recommender recommender=new GenericUserBasedRecommender(model, neighborhood, similarity);

    //得到推荐的结果,size是推荐过的数目
    recommendations=recommender.recommend(userID, size);
}
```

基于用户的推荐的实现是通过 DataModel、UserNeighborhood 和 UserSimilarity 构建推荐器 GenericUserBasedRecommender，从而实现基于用户的推荐策略。

2. 基于物品的推荐 ItemCF

基于物品的协同过滤首先需要计算物品之间的相似度，然后根据物品的相似度和用户的历史行为给目标用户生成推荐列表，其基本思想是将所有用户对某个物品的偏好作为一个向量来计算物品之间的相似度，得到物品的相似物品之后，根据用户历史的偏好预

测当前用户还没有表示偏好的物品，计算得到一个排序的物品列表作为推荐。Mahout 中基于物品的推荐 ItemCF 的实现类为 GenericItemBasedRecommender 类，该类的输入参数为 DataModel、ItemSimilarity、CandidateItemsStrategy（可选参数）和 MostSimilarItemsCandidateItemsStrategy（可选参数）。其中，UserSimilarity 用于定义两个物品之间的相似度；CandidateItemsStrategy 用于检索可能向用户推荐的所有物品；MostSimilarItemsCandidateItemsStrategy 用于检索所有相似的物品。基于物品的推荐 ItemCF 的具体过程如下。

（1）查询该用户的偏好物品集合$\{It_0, It_1, \cdots, It_m\}$；

（2）使用 MostSimilarItemsCandidateItemsStrategy（有多种策略，类似于 UserNeighborhood）获得用户偏好集合里每个 Item 最相似的其他 Item 构成的集合$\{Item_0, Item_1, Item_k\}$；

（3）对$\{Item_0, Item_1, \cdots, Item_k\}$里的每个 $Item_j$ 计算用户 $User_i$ 对该物品的偏好程度 $Perf(User_i, Item_j)$，其偏好程度的计算公式如下。

$$\mathrm{perf}(User_i, Item_j) = \frac{\sum_{l=1}^{k} \mathrm{sim}(It_l, Item_j) \times p(User_i, It_l)}{\sum_{l=1}^{k} \mathrm{sim}(It_l, Item_j)}$$

其中，$p(User_i, It_l)$是 $User_i$ 对 It_l 的偏好值。

（4）根据 $User_i$ 对每个 $Item_j$ 的偏好程度 $\mathrm{perf}(User_i, Item_j)$的数值从高到低排序，把前 N 个 Item 推荐给 $User_i$。

下面通过具体的实例来说明如何实现基于物品的推荐。基于物品的推荐的实现和基于用户的推荐实现非常类似，只是所使用的推荐器不同而已。前面介绍了基于用户的推荐 UserCF 使用的推荐器为 GenericUserBasedRecommender，而基于物品的推荐 ItemCF 所使用的推荐器为 GenericItemBasedRecommender。本实例只需要在基于用户的推荐引擎基础上修改推荐器即可，具体的实现代码如下。

```
public class ItemCFRecommender {
    //构造数据模型
    DataModel model=new FileDataModel(new File("ratings.dat"));

    //计算物品相似性
    ItemSimilarity itemsimilarity=new PearsonCorrelationSimilarity(model);

    //构建基于物品的推荐引擎
    Recommender recommender=new GenericItemBasedRecommender(model, itemsimilarity);
```

```
    //得到推荐的结果,size是推荐过的数目
    recommendations= recommender.recommend(userID, size);
}
```

基于物品的推荐与基于用户的推荐区分其实很简单,区别在于挖掘的关系是物品与物品之间(ItemCF)还是用户与用户之间的(UserCF)。

第10章 大数据应用

当前的大数据应用已十分广泛，其中以互联网领域为主导，涵盖医疗、交通、金融、教育、零售等各行各业。本章将介绍大数据在各行各业的应用案例，来说明大数据如何展开行业应用以及大数据在行业应用中的价值。

10.1 大数据应用现状及发展趋势

10.1.1 产业现状

随着大数据逐渐为越来越多的人所认知，各行各业对数据分析的广度和速度都有更高的要求，促使大数据厂商加快了对数据分析技术的研发创新。大数据分析也逐渐从结构化的历史数据分析到如社交网络、传感器数据等非结构化数据分析的技术创新。众多厂商也推出了针对大数据的解决方案，如 IBM、EMC、Oracle、SAP 等顶级厂商。然而，目前大数据的多数解决方案仍只是为大数据应用提供存储、处理、挖掘的平台技术，针对不同行业、不同场景的应用开发，仍处在极为初级的阶段。表 10.1 列出了一些重点大数据厂商大数据解决方案。

表 10.1 大数据厂商解决方案

IBM	**解决方案**	提供大数据技术、工具、一整套软件、系统和业务战略组成的完整解决方案
	大数据产品	(1) InfoSphere BigInsights、InfoSphere Streams 和 InfoSphere Warehouse，可用于处理静态数据和流动数据，用于快速分析非结构化或半结构化的海量数据； (2) 业务分析产品(Cognos，SPSS，ClarltySystem 等)； (3) 商业分析、优化顾问及研发专家提供解决方案
	应用价值	在云计算架构上整合软、硬件技术，强大全面的信息管理、数据分析软件，专业的咨询服务，为客户提供更加简易、及时的数据分析、挖掘、决策服务
Oracle	**解决方案**	为用户提供高度集成、端到端的大数据解决方案
	大数据产品	(1) 大数据一体机、Oracle Exalogic 中间件云服务器、Oracle Exadata 数据库云服务器、Oracle Exalytics 商务智能云服务器等构成的高度集成化产品组合； (2) 为以上大数据产品提供一线支持服务
	应用价值	通过软硬一体化的集成产品，为客户提供洞察数据及挖掘数据的商业价值

续表

EMC	解决方案	提供云计算开放式、分布式和集群技术处理的大数据解决方案
	大数据产品	统一的大数据分析平台 UAP，融合了 EMC Greenplum 关系数据库、EMC Greenplum HD Hadoop 发行版和 EMC Greenplum Chorus 等产品
	应用价值	充分发挥存储、管理和安全方面的优势，针对大数据提供分析工具、服务，具有强大的扩展性和开源的生态系统
SAP	解决方案	提供能够快速高效地处理海量数据的 HANA，及实时大数据平台的解决方案
	大数据产品	(1) 具有内存计算技术的 HANA； (2) SAP ERP、SAP Business One、SAP Suite on HANA 等针对行业应用的产品
	应用价值	帮助用户以便捷的方式快速获取实时信息，即时获取大数据洞察，并提高预测和规划能力
淘宝	解决方案	拥有国内最具商业价值的海量数据，为商家提供各类数据服务
	大数据产品	(1) 数据魔方平台：首个基于全站数据的数据产品，是淘宝从电子商务公司向深度数据服务公司转型的里程碑式的产品； (2) 量子恒道统计：致力于为各个电商，淘宝卖家提供精准实时的数据统计、多维的数据分析、权威的数据解决方案； (3) 淘宝指数：淘宝官方的免费的数据分享平台，用户可以窥探淘宝购物数据，了解淘宝购物趋势； (4) 阿里巴巴金融：专注于小微企业的融资服务提供商，提供阿里信用贷款
	应用价值	利用海量的交易数据可以为用户提供数据服务，帮助用户了解市场、顾客需求，从而改善自己的产品和运营策略

随着激烈的市场竞争，各企业的需求也促进了大数据解决方案产商对大数据的快速、实时处理分析技术的研发投入，使得大数据的应用方向逐渐明晰，成为企业掘金的新方向。

10.1.2 应用现状

大数据的应用现状是互联网领先，其他领域的大数据应用还在探索之中。搜索引擎是最早的互联网大数据应用，如 Google、百度等；定向广告是互联网大数据应用最主要的商业模式，如亚马逊、Facebook、腾讯等。大数据在互联网行业应用的基本特点如下。

(1) 主要以定向广告和个性化推荐为主；

(2) 简单的大数据应用已在互联网领域广泛开展，且大部分企业具备了自行实施应用的技术能力；

(3) 掌握有大量用户行为数据的互联网巨头可以较好地提供社会化服务。

大数据在其他传统行业的应用仍然处在探索阶段，如在医疗行业，美国 DNAnexus 为医疗机构和用户提供了基因数据的管理、分析和可视化能力；在能源行业，能源机构 Vestas 综合考虑温度、降水、风速、湿度和气压等因素，确定涡轮机的最佳安置地；在零售业，沃尔玛零售数据商业智能分析系统，可以了解到全球四千多家门店每天的销售情况并辅助制定相应的销售策略；在制造业，日本的小松公司根据挖掘机工作情况进行大数据分

析，从而判断下一年度的市场需求；在电信行业，西班牙电信基于大数据的“智慧足迹”产品可提供基于位置的大数据分析；在金融行业，美国证信所对海量信息进行交叉分析，推出七十余项新的增值服务。大数据的行业应用还包括农业、气象等领域。其中，热点应用领域有：社会化媒体、电子支付、内容提供、视频点播、视频监控、视频渲染、医学成像、生命科学、基因测序、移动传感器、智能电网、地球物理勘探、航空航天、高性能计算等。大数据在传统行业应用的基本特点如下。

（1）数据源主要来自企业内部、类型较少、实时要求较低；

（2）企业逐渐重视大数据，但当前应用相对简单，处于探索阶段；

（3）掌握大数据技术的企业较少，主要由 ICT 企业提供技术支持。

目前，互联网与传统产业不断整合，将催生出新的大数据创新应用机会，如金融与互联网整合的大数据应用：阿里小贷，基于对用户交易行为的大数据分析，为阿里面向中小企业实施信用贷款提供支撑；如交通与互联网的融合，德国电信利用大数据技术实施德国政府的无拥堵交通研究项目。

然而，融合创新的大数据应用案例目前比较少，应用处于起步阶段。不过，融合发展能够将互联网的在线、数据快速积累和获取等优势带至传统行业，为实体经济发展带来新的突破，将是未来大数据发展的重要方向。

10.1.3　发展趋势

大数据在越来越多的领域当中逐渐得到广泛的应用。据 2013 年的一项调查曾指出，28％的全球企业和 25％的中国企业已经开始进行大数据实践。由于各个行业都存在大数据应用需求，潜在市场空间非常可观。大数据在这些领域的主要应用如表 10.2 所示。

表 10.2　大数据在行业应用案例

行　业	案　例	商业价值
互联网	互联网广告、用户行为分析、内容推荐、个性化营销、搜索引擎优化等	改善社交网络体验、提升网络用户忠诚度、向目标用户提供有针对性的商品与服务等
金融/银行	反洗钱、反欺诈、客户价值分析、目标市场客户聚类、贷款偿还能力预测、股票等投资组合趋势分析等	降低金融风险、提高整体收入、增加市场份额等
电信行业	业务设计优化、用户行为分析、个性化推荐、用户流失预测、网络质量优化等	提高业务效率、个性化服务、优化产品套餐等
医疗卫生	临床数据比对、临床决策支持、预防传染病蔓延、就诊行为分析、疾病模式分析等	改善诊疗质量、加快诊疗速度等
公共安全	嫌疑人行为预测分析、恐怖活动检测、危险性分析、信息比对碰撞、关系人分析等	更好地对外提供公共服务、舆情分析、准确预判安全威胁等
智慧交通	面向公众的实时智能导航、实时套牌检测比对、交通违法智能挖掘与取证、机动车牌号精准查询和轨迹跟踪、基于全局的智能调度与诱导等	降低人力成本、减轻交巡警的执勤压力、轻松完成复杂的交通预判和疏导等

续表

行　业	案　例	商业价值
电力行业	电网负荷实时监控和智能调度、电力生产、用户用电状态实时监控和分析、智能电网等	海量历史/实时数据管理能力、配电/输电/变电/用电环节综合监控能力等
烟草行业	自适应的卷包排产、实时的工艺参数最优化、自推理的异常检测与故障诊断等	优化工艺参数和生产调度、在线的实时检测、设备故障预警等
制造行业	产品故障及失效综合分析等	优化产品设计、降低保修成本、降低工程事故风险等
能源行业	勘探、钻井等传感器阵列数据集中分析等	降低工程事故风险、优化勘探过程等
零售行业	基于用户位置信息的精确促销、社交网络购买行为分析等	促进客户购买热情、顺应客户购买行为习惯等

其中，互联网、电信、金融三大行业大数据应用投资占据较大的市场份额，分别达到29％、20％和17％，如图10.1所示(数据来源：赛迪顾问，2013.06)。

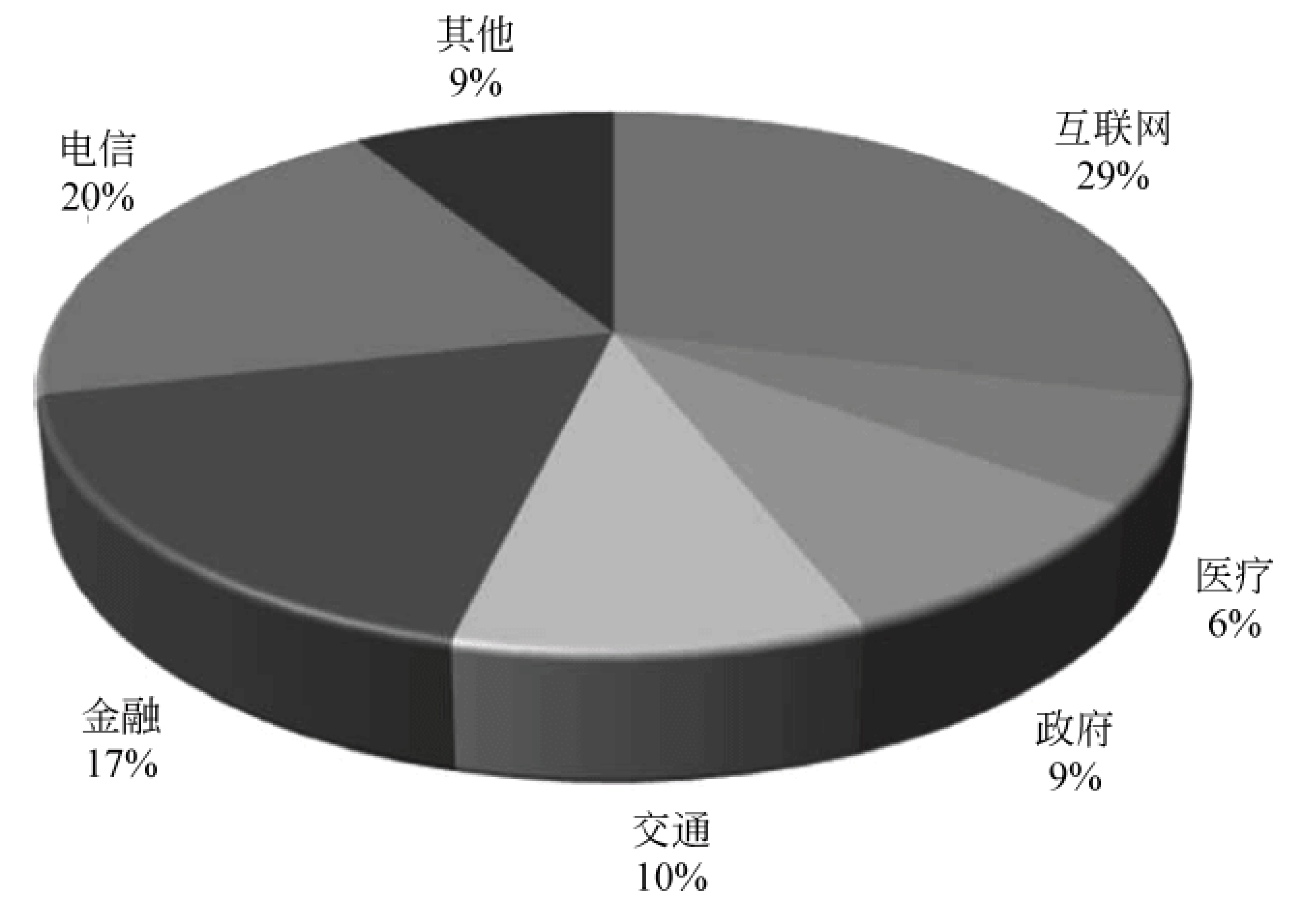


图10.1　大数据行业应用比例

互联网、电信和金融三大行业大数据未来的发展趋势将根据行业特点进行深入的大数据应用，其发展趋势如下[①]：

(1) 营销变革、信息变革及业务变革将是互联网行业大数据应用的重点。

在营销变革方面，互联网企业可以利用已拥有的海量数据资源改善其营销方式，如实时精准营销、针对性的消息推送等；在信息变革方面，互联网企业可以通过分析海量数据，反映出整体经济景气程度及各行业的变化发展状况，并能预测发展走势，为各行业提供信息支持；在业务变革方面，从多互联网巨头除了可以在相应领域建立竞争优势之外，还可以促进业务的延伸及多元化发展，并利用数据资源探求跨行业的业务融合。

(2) 服务支撑、创新支撑及运行支撑将是电信行业大数据应用的重点。

在服务支撑方面，针对个人客户，电信运营商可以利用大数据改善个人客户的服务体

① 艾瑞咨询集团.大数据行业应用展望报告，28/10/2013.



验和进行服务推荐和营销，针对企业客户，可以提出具有针对性的整合方案；在创新支撑方面，通过业务资源和账务多方面的综合分析，可以帮助电信企业进行商业决策和商业模式的创新，同时及时发现新的商业机会进行业务创新，如建立商业智能系统；在运行支撑方面，可以利用大数据提供端到端的网络质量的分析，快速对网络进行定位和修复，在提高网络质量的同时，还可以降低电信网络运营的管理成本和运维成本。此外，运营商还可以根据数据和业务的生命周期，整合新的 IT 架构和原有的架构，优化组织架构，提升内部管理能力。

(3) 客户洞察、市场洞察及运营洞察将是金融行业大数据应用重点。

在客户洞察方面，金融企业可以通过对行业客户相关的海量服务信息流数据进行捕捉及分析，以提高服务质量。同时可利用各种服务交付渠道的海量客户数据，开发新的预测分析模型，实现对客户消费行为模式进行分析，提高客户转化率；在市场洞察方面，大数据可以帮助金融企业分析历史数据，寻找其中的金融创新机会；在运营方面，大数据可协助企业提高风险透明度，加强风险的可审性和管理力度。同时也能帮助金融服务企业充分掌握业务数据的价值，降低业务成本并发掘新的盈利机会。

据预测，到 2016 年，整个市场规模将达到 30 亿左右，如图 10.2 所示(数据来源：CCW Research，1/2014)。

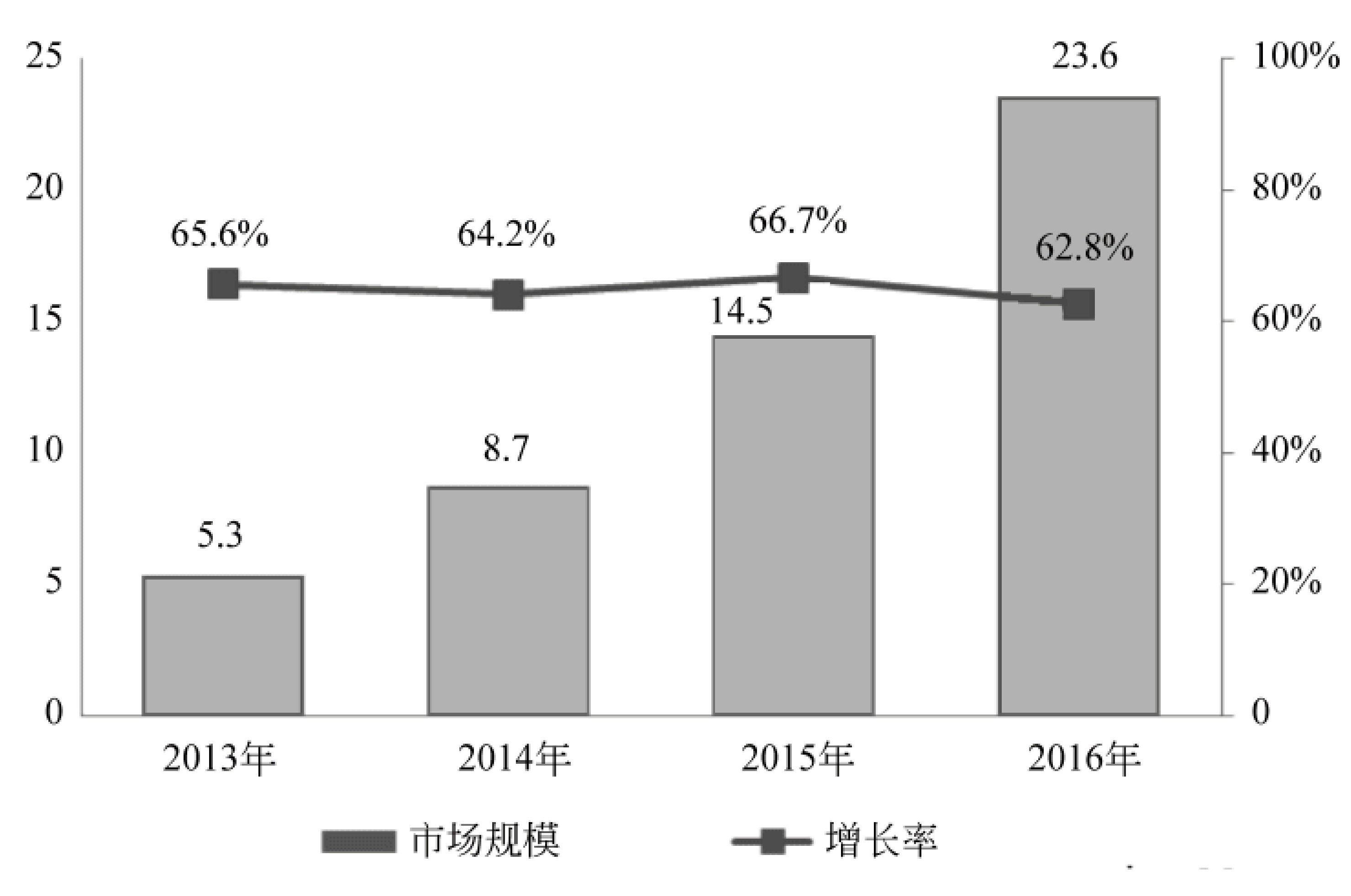


图 10.2　中国大数据市场规模

随着行业用户面临着海量数据的存储、读取、分析等多方面的需求和挑战，并结合大数据在互联网的应用经验，从总体趋势上来看，大数据未来的发展趋势将把数据资产化(拥有数据就掌控了对市场的支配和巨大的经济回报)、决策智能化(从数据中发掘洞察)，以及大数据行业应用的垂直整合的趋势(大数据行业应用的垂直整合包括从硬件到软件再到服务的行业垂直整合)；从应用方向上来看，通过对大数据的储存、分析和挖掘、大数据在互联网营销、企业管理、数据标准化与情报分析等领域将大有作为；从技术方向上来看，大数据的存储应向容量大、易扩展、更高性能、多集成、智能化、安全可靠及弹性成本等方向发展。

10.2 互联网大数据应用

大数据应用起源于互联网行业，而且互联网也是大数据技术的主要推动者。随着互联网和移动互联网快速产生的各类数据，包括用户行为数据、网页数据、系统日志数据、用户交易数据等，映射出各种大数据的创新应用，如 Facebook 对用户基本属性、行为习惯和兴趣等进行语义分析，为广告商提供基于数据挖掘的自助式广告下单服务系统；亚马逊利用大数据技术为用户提供社会化推荐、广播式个性化推荐等服务，从而加快产品的传播速度；淘宝对于大数据的应用在金融方面取得了良好的效果，在营销方面也陆续推出数据魔方、淘宝指数等数据产品；Facebook 对大量用户产品使用状况的数据进行分析，优化产品设计及服务，改善用户的使用体验等。

针对当前互联网主要面临对海量数据处理效率低、实时分析能力差、数据利用率不高等问题，并结合互联网行业大数据业务需求的特点，将互联网行业大数据总体架构分为5层，分别为数据层、存储层、计算层、模型层和应用层。其中，数据层主要实现数据采集、数据清洗和过滤等；存储层主要用于存储和管理互联网海量关系型和非关系型数据，解决各种结构化和非结构化数据的分布式采集、索引、管理、负载均衡、容灾备份等，如索引集中存储、存储访问瓶颈、关联关系计算性能、分布式查询比对等；计算层采用目前主流的批处理计算 MapReduce、实时计算 Spark 和流式计算 Storm，为互联网海量数据的处理和分析提供了计算支撑；模型层主要支持运行于分布式文件系统和分布式计算平台之上的分布式数据挖掘和数据分析算法，如逻辑斯特回归、朴素贝叶斯分类算法、*K* 均值、谱聚类算法及其分布式实现等，提供分布式挖掘算法的统一操作原语和执行引擎；应用层是按照互联网业务需求，在对数据资源的分析、查询等操作结果的基础上为用户提供服务。互联网大数据的总体架构如图 10.3 所示。



图 10.3　互联网大数据架构

如图 10.3 所示的架构是基于 Hadoop 的大数据平台之上实现了对海量用户点击数据、用户行为数据、用户日志数据进行有效的存储和管理；利用了基于 Mahout 的数据挖掘和数据分析算法，构建用户行为分析、用户聚类分析、并联分析、用户兴趣等模型；采用了 Hadoop 的批处理计算 MapReduce、实时计算 Spark、流式计算 Storm 等分布式计算技术，根据业务需求可选择实现对互联网海量数据的分析和挖掘；利用海量数据可视化、历史流展现技术、空间信息流展现技术、时间序列展现技术、社会化网络关系等大数据展现技术实现用户互联网搜索行为、消费行为、关联关系及用户趋势展现等，从而为用户提供内容推荐、个性化营销、实时精准广告推荐等应用服务。

10.3　金融行业大数据应用

目前，金融行业的信息化水平已相当高，众多金融机构都建立起了自己的数据平台，对金融行业的交易数据，如银行、证券、保险等各个金融领域的数据进行采集、存储和处理。这些数据具备大数据 4V 性的特点，而传统的数据分析手段已无法满足新业务需求，如传统数据分析无法细化到每笔交易的查询、分析，对海量数据的处理计算能力、原有数据分析速度能力不足。如何对这些海量数据进行科学的分析处理、挖掘隐藏在数据内部各种有价值的关联及提供决策支持，成为金融行业面临的新挑战。

由于金融行业的很多业务系统都已经建成，这就需要金融行业的大数据应用的解决方案能全面整合金融数据，在客户深度分析、反省钱、反欺诈预警等方面充分发挥出金融大数据的价值，保证金融行业海量交易数据的安全性、可靠性和高效率的运营。这里我们需要设计一种可嵌入式的金融行业大数据平台，在不影响现有业务系统正常运作的前提下，将各业务系统的数据进行数据整合与共享。金融行业的可嵌入式大数据总体架构的设计仍可分为 5 层，分别为数据层、存储层、计算层、模型层和应用层，如图 10.4 所示。

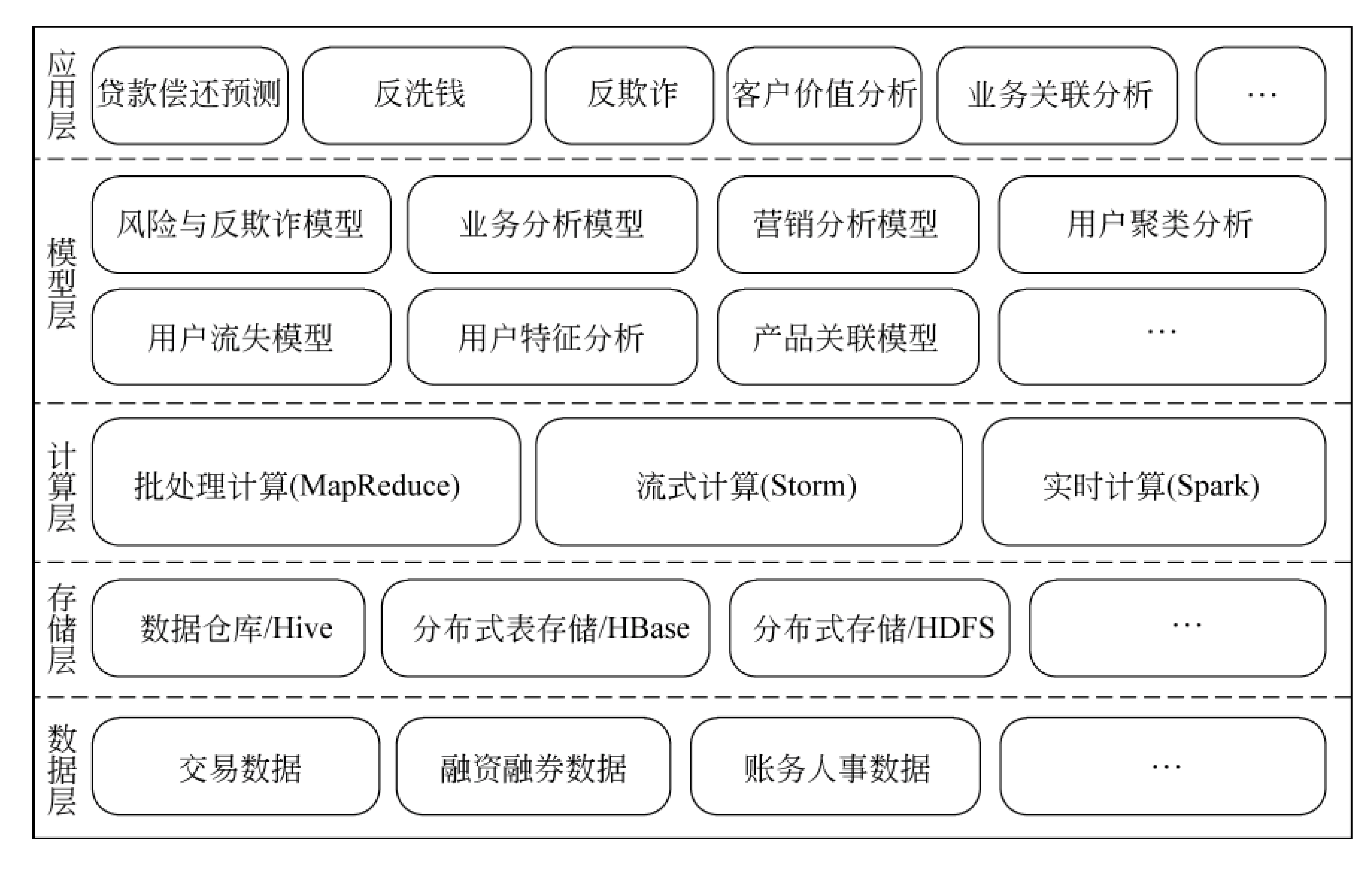


图 10.4　金融行业大数据架构

如图 10.4 所示的架构是基于 Hadoop 的大数据平台之上实现了对海量金融交易、融资融券数据等进行有效的存储和管理;在模型层利用了数据挖掘和数据分析算法,构建风险与反欺诈、业务分析、营销分析、用户特征分析、产品关联等模型;在计算层采用了 Hadoop 的批处理计算 MapReduce、实时计算 Spark、流式计算 Storm 等分布式计算技术,根据业务需求可选择不同的计算引擎实现业务数据的分析和挖掘;利用海量数据可视化、历史流展现技术、空间信息流展现技术、时间序列展现技术等大数据展现技术提高金融数据的直观性和可视性,从而提升金融数据的可利用价值。

10.4 电信行业大数据应用

随着移动互联网、云计算等新技术的兴起,传统的电信运营商已从传统的语言业务结合少量的数据业务转型为以数据业务为主体的业务模式,从而促使电信行业的业务系统呈现了新的业务形态和数据类型。面对海量的结构和非结构化数据对电信行业所带来的挑战已不再局限于存储和传输的问题,更重要的是如何做好海量结构化和非结构化数据的分析,从而更好地服务客户、提高业务效率等。

针对当前电信行业主要面临对海量数据处理效率低、数据利用率不高等问题,并结合电信行业大数据业务需求的特点,引入电信行业大数据,实现对海量的文本、图片、语音、视频等非结构化数据和业务系统产生的结构化数据的存储、管理、分析挖掘。我们将电信行业大数据应用的总体架构仍分为 5 层,分别为数据层、存储层、计算层、模型层和应用层,如图 10.5 所示。

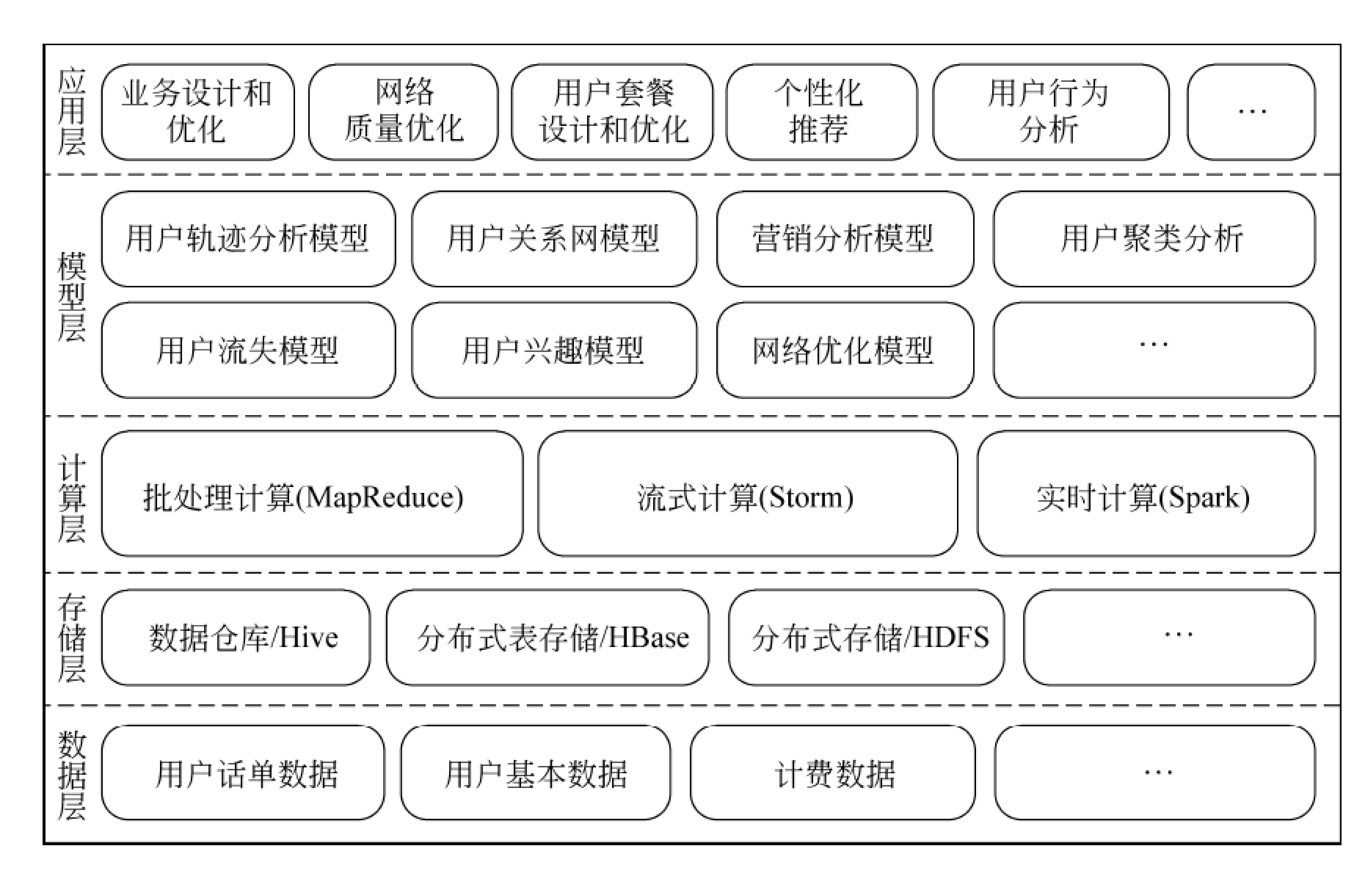


图 10.5 电信行业大数据架构

如图 10.5 所示的架构是基于 Hadoop 的大数据平台之上实现了对海量用户话单数据、用户行为数据、用户基本数据等进行有效的存储和管理；利用数据挖掘和数据分析算法，构建网络优化、用户聚类分析、用户关系网、用户兴趣等模型；采用了 Hadoop 的批处理计算 MapReduce、实时计算 Spark、流式计算 Storm 等分布式计算技术，根据业务需求可选择实现对电信行业海量数据的分析和挖掘；利用海量数据可视化、历史流展现技术、空间信息流展现技术、时间序列展现技术、社会化网络关系等大数据展现技术实现流量经营、网络优化、数据服务、个性化服务等。

10.5 医疗行业大数据应用

随着医疗技术的发展，医疗行业积累了大量不同类型的数据，如健康档案、电子病历、医学图像等海量数据，已成为医疗行业宝贵的财富。这些数据存在着数据量庞大、数据类型复杂多变、数据利用率低等问题，而且随着健康医疗的不断发展，非结构化数据增速将持续加快，传统的关系型数据库在存储大数据集时失去性能、功能和成本优势，并且在处理和查询大数据集时更是力不从心。如果能够对这些数据进行有效的存储、处理、查询和分析，就可以帮助临床医生做出更为科学准确的诊断、用药决策、病理分析、人体健康度分析、个性化差异分析等。

针对当前医疗行业主要面临数据量增长速度快、数据类型复杂多变、对海量数据处理效率低、数据利用率不高等问题，结合医疗行业大数据业务需求的特点，引入医疗行业大数据，有助于健康档案数据的管理和服务、基础医疗服务、临床决策支持、疾病模式分析、个性化医疗等。我们将医疗行业大数据应用的总体架构仍分为 5 层，分别为数据层、存储层、计算层、模型层和应用层，如图 10.6 所示。



图 10.6 医疗行业大数据架构

如图 10.6 所示的架构是基于 Hadoop 的大数据平台之上实现了对用户病历数据、临床数据、用户就诊数据等进行有效的存储和管理；在模型层利用了数据挖掘和数据分析算法，构建临床分析、用户行为分析、疾病特征分析、病毒扩散、基因分析等模型；在计算层采用了 Hadoop 的批处理计算 MapReduce、实时计算 Spark、流式计算 Storm 等分布式计算技术，根据业务需求可选择不同的计算引擎实现业务数据的分析和挖掘；利用海量数据可视化、历史流展现技术、空间信息流展现技术、时间序列展现技术等大数据展现技术提高医疗行业数据的直观性和可视性，从而提升医疗行业数据的可利用价值。

10.6 智慧交通大数据应用

大数据下的智慧交通就是整合传感器、监控视频和 GPS 等设备产生的海量数据，并结合气象监测设备产生的天气状况数据、人口分布数据、移动通信数据等相结合，从这些数据中洞察出我们真正需要的有价值信息，从而实现智能交通公共信息服务的实时传递和快速反应的应急指挥、智能交通业务联动快速应对、可视化事件跟踪、套牌实时监测预警等。然而，面对每天近千万辆轿车、轨道交通、快速公交系统，以及高并发事件、实时数据流的处理需求，对海量非结构化数据的组织与分析需求等，成为智慧交通面临的重大挑战。

基于大数据的智慧交通可以有效地管理交通数据，如可集中访问分散存储在不同支队数据中心的图像或视频等交通数据；提高对海量数据的利用，如可从海量数据中挖掘出有价值的信息为公安治安、刑侦、经侦等部门人员及一线民警提供信息支撑服务；改善交通，如提高对各种交通突发事件的应急调度能力，依据历史数据预测交通或突发事件的趋势。由于交通行业已经过多年的信息化发展，智慧交通大数据应用的总体架构可分别为感知层、传输层、数据层、存储层、计算层、模型层、应用层。其中，感知层主要由传感器、雷达接收器、RFID 阅读器、感应处理器、红外接收器等组成，传输层主要功能是将前端采集的数据传输到智慧交通大数据平台上，所以从严格意义上讲感知层和传输层并不真正属于智慧交通大数据架构的范畴，但为了智慧交通大数据解决方案的完整性，将感知层和传输层加入到大数据架构中。智慧交通大数据的总体架构如图 10.7 所示。

基于 Hadoop 的智慧交通大数据平台可有效地存储和管理交通数据，并可通过模型层对这些数据进行有效的分析，从交通数据中挖掘出潜在的交通风险，把事后处理转换为事前预警的工作模式，为公安、交通等各行各业提供更为有效的业务信息支持。

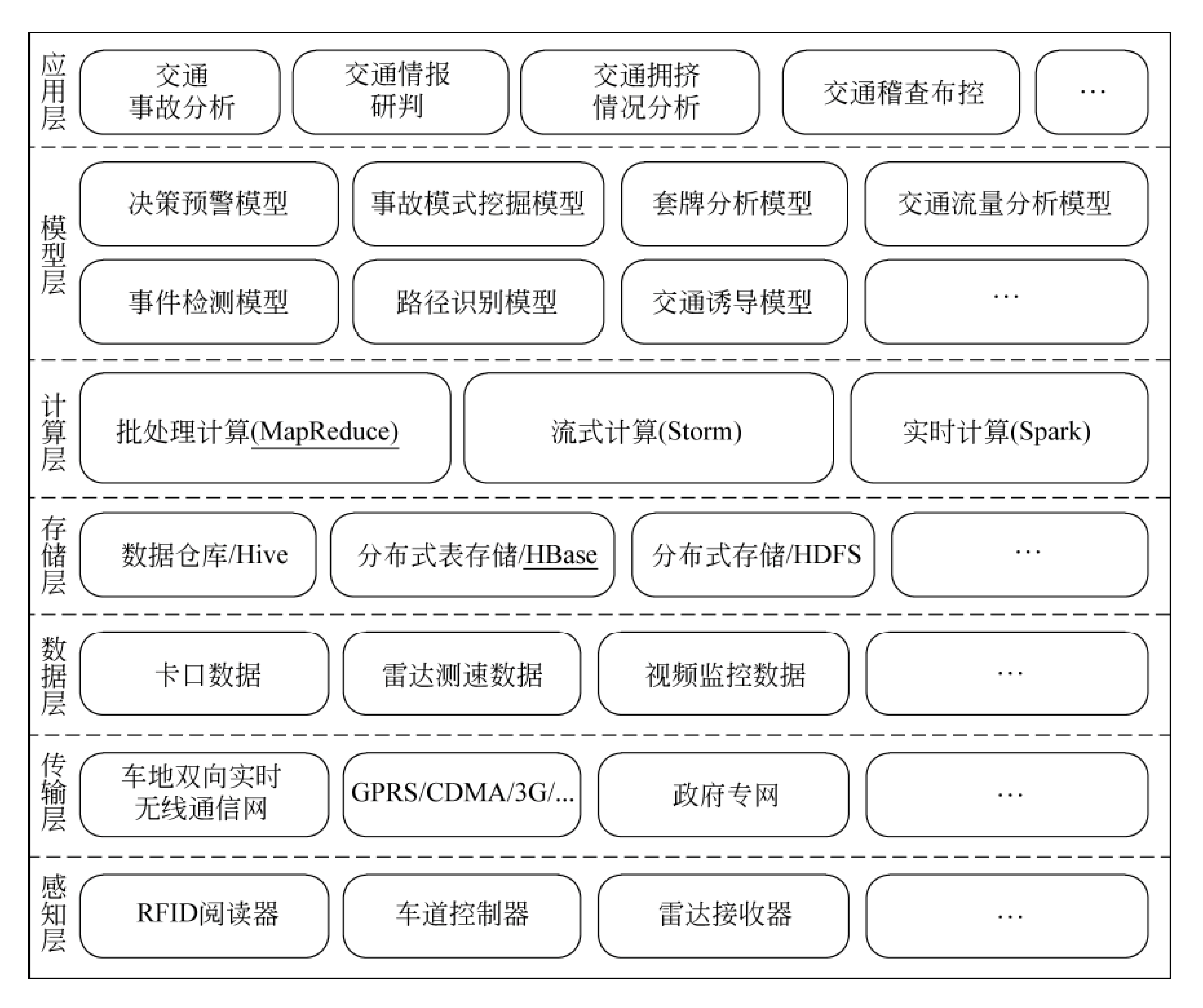


图 10.7　智慧交通大数据架构

10.7　大数据应用案例

从不同行业的大数据架构可以看出，基于 Hadoop 的大数据行业应用的区别在于数据层数据源和数据类型的不同；存储层和计算层根据实际业务需求不同选择不同的存储介质和计算引擎；模型层根据业务需求不同，根据数据挖掘和机器学习算法实现各种类型的模型；应用层根据业务应用特点实现不同的应用。其中，模型层是整个大数据的核心部分，也是区别于不同行业应用的最核心部分。模型层为上层应用提供了数据支撑，应用层是区别于不同行业应用的外在表现。本节将列举在互联网行业和智慧交通行业大数据应用的具体案例。

10.7.1　互联网大数据应用案例

用户在互联网产生了海量的用户日志行为数据，通过数据挖掘的方法可以挖掘出有用的信息。本实例将以用户日志数据为基础，通过大数据进行用户行为分析，如针对用户在购物网站访问网页的信息，可以挖掘出用户的兴趣点，从而进行物品推荐。由于互联网用户日志数量级在急剧扩充，传统的单机版分析程序已无法胜任海量日志分析的需求，因此，基于 Hadoop 的大数据分析平台成为首要考虑的技术框架。

1. 技术路线选择

基于图 10.3 的互联网大数据架构，在数据层方面主要是针对互联网用户产生的日志数据为主要处理和分析对象；在数据存储层方面主要选用分布式存储 HDFS（用于存储海量日志数据）、分布式表存储 HBase（建立海量数据的列式查询）和数据仓库 Hive（用于对海量数据进行分析挖掘）；在计算层主要选用批处理计算 MapReduce 模式（日志分析并没有很高的实时性要求）；在模型层需要为用户访问的网页行为进行建模；在应用层实现对用户兴趣挖掘的实现。具体的部署实现如图 10.8 所示。

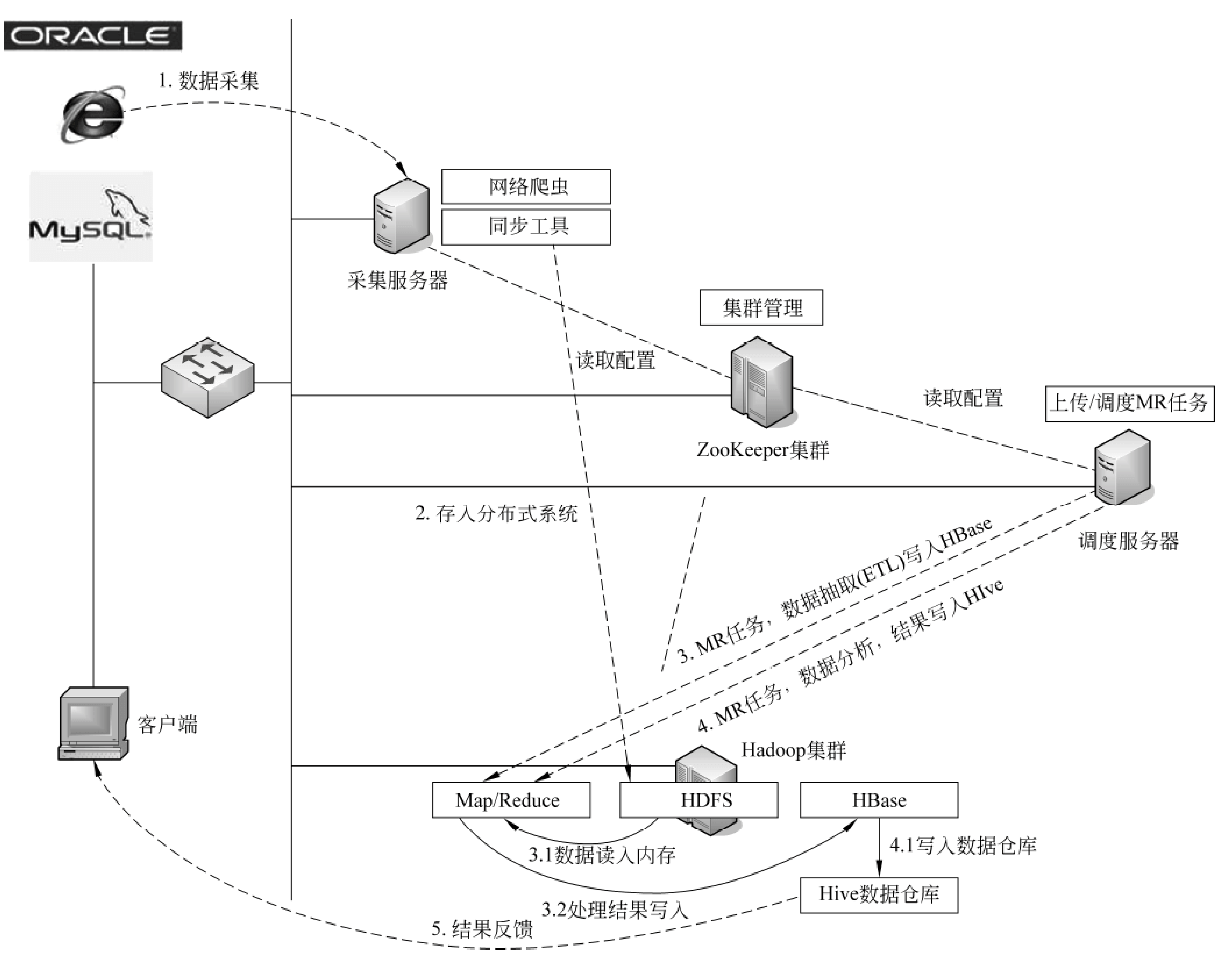


图 10.8　日志分析的部署实现

从图 10.8 可以看出，基于 Hadoop 的用户日志分析应用首先从互联网通过采集服务器（这里可用 Flume 或 Kafka 等数据采集工具）采集用户日志数据或从已有的数据库（Oracle 或 MySQL 等）进行数据同步（这里可用 Sqoop 数据同步工具）。然后，将采用的数据存入分布式文件系统 HDFS 中，通过批处理计算 MapReduce 进行数据抽取、分析等 ETL 工作。将 ETL 后的数据写入分布式列式数据库 HBase，以便进行查询或为下一步的分析做好准备工作。最后，在 HBase 中选择要进行分析的数据集合写入到数据仓库 Hive 中，通过模型层的数据挖掘算法来对 Hive 中的数据进行数据分析和挖掘，并将结果

反馈给客户端。整个 Hadoop 集群由集群管理服务器 ZooKeeper 和任务调度服务器 Oozie 来负责整个集群的管理和 MapReduce 作业的调度工作。

2. 用户行为建模

日志分析应用案例在数据层和存储层的相关细节请结合前面章节的知识点自行解决，这里将不再赘述。本节将重点放在如何通过用户行为建模实现海量日志分析应用。用户在互联网访问网页时，一般都会在系统日志中存储一条记录（用户＋URL＋访问时间），这些网页记录是推断用户行为的基础，用户行为建模过程如下。

（1）根据 URL 得到网页内容信息，并对网页内容通过机器学习算法进行处理，从而得到代表此网页的几个关键词；

（2）对用户访问关键词信息进行汇总，得到用户关注的关键词列表，并对每个关键词在 URL 中出现的次数来赋予不同的权重；

（3）对用户关注的关键词列表进行一定的扩展或归约操作，从而得到更加具有意义的关键词信息，更好地表达用户的兴趣点。

根据用户行为建模的流程，将各个模块进行并行化 MapReduce 实现。其中整个应用的输入是用户访问网页记录组成的日志文件，每行表示用户访问网页的一条记录，其形式为：

```
用户　URL
```

期望输出为用户的兴趣点文件，文件每行存储每个用户的兴趣点，其形式为：

```
用户　词 1 权重 1 词 2 权重 2 词 3 权重 3…
```

对于关键词权重的衡量标准可以把该词在单一网页中出现的次数 TF 和在所有网页中出现的次数 DF 作为参考标准，通过机器学习算法计算出其关键词的权重。首先通过 MapReduce 来计算关键词在单一网页中出现的次数 TF，MapReduce 设计如下。

1）Map

Map 的输入为＜用户，URL＞列表，对于单条记录，进行 URL 爬虫和分词，得到用户访问的该网页中包含所有词的信息。当遇到一个词，Map 进行一次输出，key 为用户＋网页＋词，value 为 1。还可以统计该词的其余特征值，如词性等。这里为方便演示就不再增加其他特征，读者可自行添加。

2）Reduce

Reduce 是将 Map 输出结果中相同的 key 汇聚到一起，并将用户＋网页＋词仍作为 key 进行输出，将每组中记录条数作为 value 进行输出。这样 Reduce 的输出结果文件对应记录为：用户＋URL＋TF。

为了提高处理效率，可以在 Map 阶段完成每个词的 TF 统计，从而减少了 Map 和 Reduce 之间大量数据传输的时间消耗，具体的做法是采用数据结构 hashMap＜String，value＞，Map 的输入不变，只是执行一次 Map 操作的结果加入到 hashMap 中，其中 key

为词，value 变为原来 value 值+1。这样就可将 Reduce 阶段省略，具体实现代码如下。

```
public class TF extends Configured implements Tool, Mapper< Text, Text, Text, IntWritable> {
  public void map(Text usr, Text url, OutputCollector< Text, IntWritable> output, Reporter reporter)
  throws IOException {
    //HashMap统计 TF
    HashMap< Text, int> wordCount= new HashMap< Text, int> ();
        Text[] words= callCrawl(url);                    //调用爬虫程序
        for(Text word: words){                           //统计词次数信息
            int count= wordCount.get(word);
            wordCount.put(word, (count> 0)? (count+ 1):1);
        }
        Iterator< Text,int> iter= wordCount.entrySet().iterator();
        while(iter.hasNext()){
            Map.Entry< Text, int> entry= iter.next();
          //Map输出,key为用户+url+词,value为 TF
           output.collect(usr+ url+ entry.getKey(), entry.getValue());
        }
    }
    public void runCal(Path input, Path output)throws IOException {
      JobConf job= new JobConf(getConf(), TFCal.class);
      //设置 InputPath, outPath, MapperClass, InputFormat, OutputFormat, …
      job.setReduceNum(0);                                  //Reduce数目设为 0,不进行 Reduce操作
      JobClient.runJob(job);
    }
}
```

然后，通过 MapReduce 来计算关键词在所有网页中出现的次数 DF。在 DF 信息基础上再通过 MapReduce 来计算每个词的权重，即该词成为网页关键词的概率，此处可以设定一个阈值，如果该词成为关键词的概率大于阈值，表示该词可以成为该网页的关键词，将其输出，否则忽略。MapReduce 的具体设计如下。

1) Map

Map 的输入为 TF 计算阶段的输出，key 为用户+网页+词，value 为词的 TF。该 Map 阶段将除去网页信息，其输出 key 为用户+词，输出的 value 为 1。

2) Reduce

Reduce 以 Map 输出作为输入，用户访问词相同的会被同一个 Reduce 进行处理，Reduce 中统计该组包含记录的数目，即为该关键词的 DF。这里为方便查询到关键词 DF 的信息，将 Reduce 阶段的输出以索引文件的形式进行输出(使用 Lucene 建立索引的机制)，Reduce 的输出 key 为用户+词，value 为 DF 值。

通过 MapReduce 来计算关键词在所有网页中出现的次数 DF 及每个词的权重的具体实现代码如下。

```
public class DF extends Configured implements Tool, Mapper<Text, IntWritable, Text, IntWritable>,
Reducer<Text, IntWritable, Text, LuceneDocumentWrapper>{
    public void map(Text key, IntWritable url, OutputCollector<Text, IntWritable> output, Reporter
    reporter)throws IOException{
        //将 key 拆分成 user,url,word 三部分
        output.collect(user+word, new IntWritable(1));
    }

    public void reduce(Text key, Iterator<IntWritable> iter, OutputCollector
    <Text, LuceneDocumentWrapper> output, Reporter reporter)throws IOException{
   int df=iter 中包含元素数目;
   //建立 Lucene 索引,以用户+词为 key,DF 作为 value,进行存储
   Document doc=new Document();
   doc.add(new Field("word", key.toString(), Field.Store.NO,Field.Index.UN_TOKENIZED));
   doc.add(new Field("DF", df, Field.Store.YES,Field.Index.NO));
   output.collect(new Text(), new LuceneDocumentWrapper(doc));
    }

    public void runDF(Path input, Path output)throws IOException{
        JobConf job=new JobConf(getConf(), DFCal.class);
        //设置 InputPath, outPath, MapperClass, InputFormat, …
        //设置输出格式为 LuceneOutputFormat
        job.setOutputFormat(LuceneOutputFormat);
        JobClient.runJob(job);
        //合并各个 Reduce 阶段生成的索引文件为一个完整索引文件
        //实现 Lucene 的 IndexWriter 类
    }
    …
 }
//计算每个词的权重
public class KeyWord extends Configured implements Tool, Mapper<Text, Text, Text, IntWritable>{
    String fWeights[];                              //记录特征权重
    IndexSearcher searcher=null;                    //用于查询 Lucene 索引文件
    public void map(Text key, Text wordInfo, OutputCollector<Text, IntWritable>
    output, Reporter reporter)throws IOException{
    //解析 key,从中得到词信息,再查找索引文件,得到 DF
    Term t=new Term("word", word);
    Query query=new TermQuery(t);
    Hits hits=searcher.search(query);
    if(hits.length()==1){
      Document doc=hits.doc(0);
```

```
            String df=doc.get("DF");
            //从词信息中提取出来每个特征对应取值，存储在数组 val 中
            weight=sum(val[i] × fWeights[i]);          //计算该词作为关键词权重
            if(weight >= threshold)                    //权重大于阈值的视为网页关键词
            //关键词输出,key 包含用户+关键词,value 为 1
          output.collect(key, new Writable(1));
                }
          }

      public void configure(JobConf job){
      String fWeightPath=job.getStrings("fWeight.path");          //内部获得特征权重路径
      //读取特征权重文件,得到特征权重列表,填入 fWeights
      String dfPath=job.getStrings("DF.path");
      FsDirectory fsDirectory=new FsDirectory(FileSystem.get(getConf()),dfpath, false, getConf());
      searcher=new IndexSearcher(fsDirectory);
        }
        public void runkeyWord(String input, String output, String DFPath){
            String featureWeightFile;
            MachineLearning(featureWeightFile);        //调用机器学习算法,计算特征权重
            JobConf job=new JobConf(getConf(),KeyWord.class);
            //设置参数,以传入 Map 和 configure
            job.setStrings("fWeight.path", featureWeightFile);
            job.setStrings("DF.path", DFPath);          //设置 DF 索引文件位置
            JobClient.run(job);
        }
        …
  }
```

最后，在获得了关键词列表的基础上进行扩展，扩展的思想有很多种，这里就简单介绍其中一种扩展方法：对于关键词列表中的每个词 A，通过查找得到与词 A 相关的词列表 A_L。遍历 A_L，如果 A_L 中的词 B 也被用户访问过，那么要将用户访问词 B 的值×A 和 B 的相关度的结果加入到 A 的旧值上；如果 A_L 中的词 C 用户没有访问过，则要把词 C 加入到用户访问词列表中，并将词 A 的旧值×A 和 C 的相关度加入到词 C 值上。该过程只需要一个 Map 过程即可完成操作。Map 以用户访问的词列表作为输入，key 为用户，value 为关键词列表，Map 的输出 key 仍为用户，value 为用户访问的词列表及对应的新的关注度值，具体的代码实现如下。

```
public class WordExp extends Configured implements Tool, Mapper< Text, Text, Text, Text >{
    IndexSearcher searcher=null;                  //用于查询 Lucene 索引文件
    public void map(Text key, Text val, OutputCollector< Text,Text> output,Reporter reporter)throws
IOException {
```

```
HashMap<String, float>words;                    //key 为词,value 为用户访问该词的权重
HashMap<String, float>wordNewInfo;              //存储调整后的列表信息
//将 val 关键词信息进行解析,依次置入 words;
//复制 words 中信息至 wordNewInfo 中 ;
    for(words 中每一个关键词 word){
    float w1=words.get(word);
    //查找 Lucene 索引文件,得到该词相关词列表 corrWords;
    for(corrWords 中每个词 corrW){
    //如果 corrW 也被用户访问,修改两个词的权重
    if((float w2=words.get(corrW))!=null){
    wordsNewInfo.put(word, wordsNewInfo.get(word)+w2 * corrW.pij);
    wordsNewInfo.put(corrW, wordsNewInfo.get(corrW)+w1 * corrW.pij);
            }else{                              //如果未被访问,将词加入到用户访问列表中
                wordsNewInfo.put(corrW, w1 * corrW.pij);
            }
        }
    }
    String wordListNew="";
    for(wordNewInfo 中每个元组 entry)
        wordListNew=wordListNew+entry.getKey()+entry.getVal();
    output.collect(key, new Text(wordListNew);
}
…
}
```

到此,就完成了用户行为分析建模。该模型在应用层可实现精准的广告营销、内容推荐、用户行为分析等应用。例如,在内容推荐或实时精准广告营销方面,如果根据应用关注的网页中提取出的关键词是“足球”、“篮球”、“游泳”等,而这些词都被划分为体育类,通过用户行为模型,可以推测出该用户对“体育”比较感兴趣,则可以多推荐体育方面的内容或体育类的产品。有关应用层更多的应用,读者需要根据实际的业务需求来设计应用场景。

10.7.2　智慧交通大数据应用案例

随着机动车辆不断增加、交通运输也越来越繁忙,交通数据以每年 60% 的速度不断增长,这些数据被分散存储在不同的市、区、县级的数据中心,跨数据中心使用这些数据非常不便,而且依靠传统的手工方式实现数据查询和统计分析的效率随着数据量的增加变得越来越低,对数据的利用率不高。

基于图 10.7 的智慧交通大数据架构,在感知层和传输层方面主要是将采集的交通数据传输到数据层;在数据层方面主要是将这些分散的数据进行资源整合,为下一步的数据处理和分析提供必要保障;在存储层方面主要选用分布式存储 HDFS(用于存储海量日志数据)、分布式表存储 HBase(建立海量数据的列式查询)和数据仓库 Hive(用于对海量数

据进行分析挖掘);在计算层主要选用批处理计算 MapReduce 模式(用于对实时性要求不高的数据处理业务)和实时计算 Spark 模式(用于实时流的数据处理);在模型层需要为车辆轨迹分析、套牌分析等应用提供数据分析模型;在应用层实现以数据为驱动的道路通行实时状况分析、交通事故统计分析等应用,具体的部署实现如图 10.9 所示。

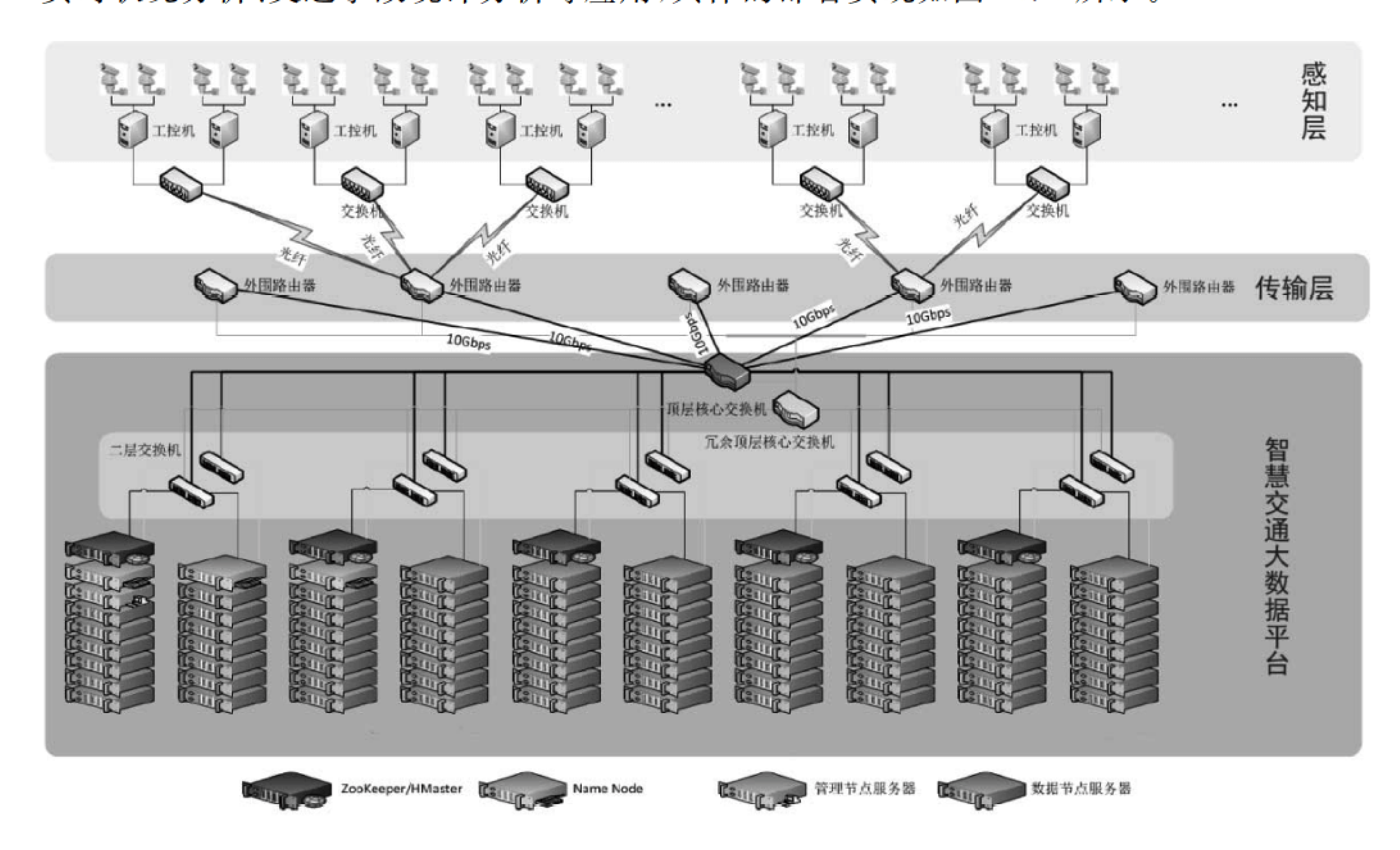


图 10.9　智慧交通大数据部署实现

其中,智慧交通大数据平台中所选择的技术路线的软件架构如图 10.10 所示。

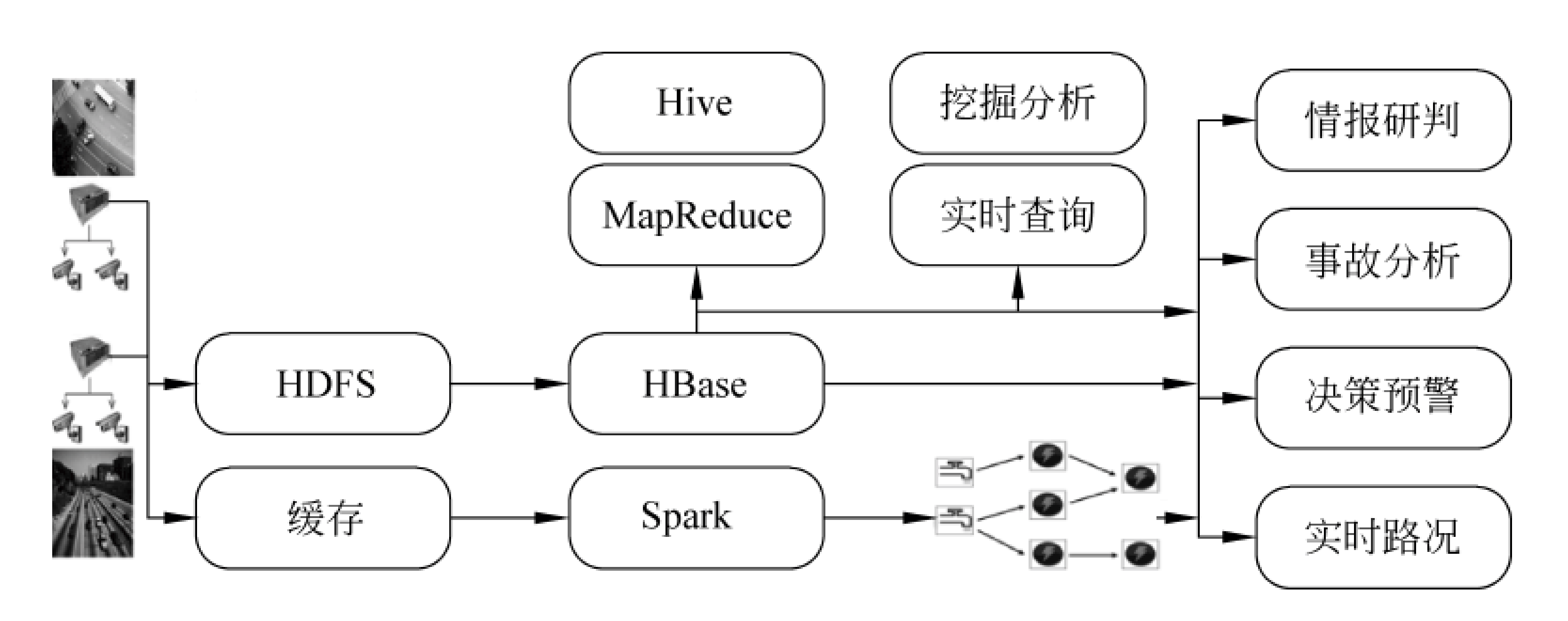


图 10.10　智慧交通大数据平台软件架构

采用基于 Hadoop 的智慧大数据平台可解决海量交通数据存储和管理、无缝容量扩充,利用大数据分析技术实现对交通数据的智能、高效的处理和分析,提高了车辆轨迹分析、套牌分析、车辆查缉布控等系统的查询性能及实时性,充分挖掘数据背后的隐藏价值。

附　　表

附表 1　Hadoop 默认端口及作用

端　口	作　　用
8030	yarn. resourcemanager. scheduler. address
8031	yarn. resourcemanager. resource-tracker. address
8032	yarn. resourcemanager. address
8033	yarn. resourcemanager. admin. address
8040	yarn. nodemanager. localizer. address
8042	yarn. nodemanager. webapp. address
8088	yarn. resourcemanager. webapp. address
8090	yarn. resourcemanager. webapp. https. address
8188	yarn. timeline-service. webapp. address
8480	dfs. journalnode. rpc-address
8481	dfs. journalnode. https-address
9000	fs. defaultFS
9001	dfs. namenode. rpc-address
10020	mapreduce. jobhistory. address
19888	mapreduce. jobhistory. webapp. address
50010	dfs. datanode. address
50020	dfs. datanode. ipc. address
50070	dfs. namenode. http-address
50075	dfs. datanode. http. address
50090	dfs. namenode. secondary. http-address
50091	dfs. namenode. secondary. https-address
50100	dfs. namenode. backup. address
50105	dfs. namenode. backup. http-address
50470	dfs. namenode. https-address
50475	dfs. datanode. https. address

附表 2　Hadoop 1.0 与 Hadoop 2.0 属性名称变化对比

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
create. empty. dirif. nonexist	mapreduce. jobcontrol. createdir. ifnotexist
dfs. access. time. precision	dfs. namenode. accesstime. precision
dfs. backup. address	dfs. namenode. backup. address
dfs. backup. http. address	dfs. namenode. backup. http-address
dfs. balance. bandwidthPerSec	dfs. datanode. balance. bandwidthPerSec
dfs. block. size	dfs. blocksize
dfs. data. dir	dfs. datanode. data. dir
dfs. datanode. max. xcievers	dfs. datanode. max. transfer. threads
dfs. df. interval	fs. df. interval
dfs. replication. min	dfs. namenode. replication. min
dfs. replication. pending. timeout. sec	dfs. namenode. replication. pending. timeout-sec
dfs. safemode. extension	dfs. namenode. safemode. extension
dfs. safemode. threshold. pct	dfs. namenode. safemode. threshold-pct
dfs. secondary. http. address	dfs. namenode. secondary. http-address
dfs. socket. timeout	dfs. client. socket-timeout
dfs. umaskmode	fs. permissions. umask-mode
dfs. write. packet. size	dfs. client-write-packet-size
fs. checkpoint. dir	dfs. namenode. checkpoint. dir
fs. checkpoint. edits. dir	dfs. namenode. checkpoint. edits. dir
fs. checkpoint. period	dfs. namenode. checkpoint. period
fs. default. name	fs. defaultFS
hadoop. configured. node. mapping	net. topology. configured. node. mapping
hadoop. job. history. location	mapreduce. jobtracker. jobhistory. location
hadoop. native. lib	io. native. lib. available
hadoop. net. static. resolutions	mapreduce. tasktracker. net. static. resolutions
hadoop. pipes. command-file. keep	mapreduce. pipes. commandfile. preserve
hadoop. pipes. executable. interpretor	mapreduce. pipes. executable. interpretor
hadoop. pipes. executable	mapreduce. pipes. executable
hadoop. pipes. java. mapper	mapreduce. pipes. isjavamapper
hadoop. pipes. java. recordreader	mapreduce. pipes. isjavarecordreader
hadoop. pipes. java. recordwriter	mapreduce. pipes. isjavarecordwriter

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
hadoop. pipes. java. reducer	mapreduce. pipes. isjavareducer
hadoop. pipes. partitioner	mapreduce. pipes. partitioner
heartbeat. recheck. interval	dfs. namenode. heartbeat. recheck-interval
io. bytes. per. checksum	dfs. bytes-per-checksum
io. sort. factor	mapreduce. task. io. sort. factor
io. sort. mb	mapreduce. task. io. sort. mb
io. sort. spill. percent	mapreduce. map. sort. spill. percent
jobclient. completion. poll. interval	mapreduce. client. completion. pollinterval
jobclient. output. filter	mapreduce. client. output. filter
jobclient. progress. monitor. poll. interval	mapreduce. client. progressmonitor. pollinterval
job. end. notification. url	mapreduce. job. end-notification. url
job. end. retry. attempts	mapreduce. job. end-notification. retry. attempts
job. end. retry. interval	mapreduce. job. end-notification. retry. interval
job. local. dir	mapreduce. job. local. dir
keep. failed. task. files	mapreduce. task. files. preserve. failedtasks
keep. task. files. pattern	mapreduce. task. files. preserve. filepattern
key. value. separator. in. input. line	mapreduce. input. keyvaluelinerecordreader. key. value. separator
local. cache. size	mapreduce. tasktracker. cache. local. size
map. input. file	mapreduce. map. input. file
map. input. length	mapreduce. map. input. length
map. input. start	mapreduce. map. input. start
map. output. key. field. separator	mapreduce. map. output. key. field. separator
map. output. key. value. fields. spec	mapreduce. fieldsel. map. output. key. value. fields. spec
mapred. acls. enabled	mapreduce. cluster. acls. enabled
mapred. binary. partitioner. left. offset	mapreduce. partition. binarypartitioner. left. offset
mapred. binary. partitioner. right. offset	mapreduce. partition. binarypartitioner. right. offset
mapred. cache. archives	mapreduce. job. cache. archives
mapred. cache. archives. timestamps	mapreduce. job. cache. archives. timestamps
mapred. cache. files	mapreduce. job. cache. files

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
mapred. cache. files. timestamps	mapreduce. job. cache. files. timestamps
mapred. cache. localArchives	mapreduce. job. cache. local. archives
mapred. cache. localFiles	mapreduce. job. cache. local. files
mapred. child. tmp	mapreduce. task. tmp. dir
mapred. cluster. average. blacklist. threshold	mapreduce. jobtracker. blacklist. average. threshold
mapred. cluster. map. memory. mb	mapreduce. cluster. mapmemory. mb
mapred. cluster. max. map. memory. mb	mapreduce. jobtracker. maxmapmemory. mb
mapred. cluster. max. reduce. memory. mb	mapreduce. jobtracker. maxreducememory. mb
mapred. cluster. reduce. memory. mb	mapreduce. cluster. reducememory. mb
mapred. committer. job. setup. cleanup. needed	mapreduce. job. committer. setup. cleanup. needed
mapred. compress. map. output	mapreduce. map. output. compress
mapred. data. field. separator	mapreduce. fieldsel. data. field. separator
mapred. debug. out. lines	mapreduce. task. debugout. lines
mapred. healthChecker. interval	mapreduce. tasktracker. healthchecker. interval
mapred. healthChecker. script. args	mapreduce. tasktracker. healthchecker. script. args
mapred. healthChecker. script. path	mapreduce. tasktracker. healthchecker. script. path
mapred. healthChecker. script. timeout	mapreduce. tasktracker. healthchecker. script. timeout
mapred. heartbeats. in. second	mapreduce. jobtracker. heartbeats. in. second
mapred. hosts. exclude	mapreduce. jobtracker. hosts. exclude. filename
mapred. hosts	mapreduce. jobtracker. hosts. filename
mapred. inmem. merge. threshold	mapreduce. reduce. merge. inmem. threshold
mapred. input. dir. formats	mapreduce. input. multipleinputs. dir. formats
mapred. input. dir. mappers	mapreduce. input. multipleinputs. dir. mappers
mapred. input. dir	mapreduce. input. fileinputformat. inputdir
mapred. input. pathFilter. class	mapreduce. input. pathFilter. class
mapred. jar	mapreduce. job. jar
mapred. job. classpath. archives	mapreduce. job. classpath. archives
mapred. job. classpath. files	mapreduce. job. classpath. files
mapred. job. id	mapreduce. job. id
mapred. jobinit. threads	mapreduce. jobtracker. jobinit. threads

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
mapred. job. map. memory. mb	mapreduce. map. memory. mb
mapred. job. name	mapreduce. job. name
mapred. job. priority	mapreduce. job. priority
mapred. job. queue. name	mapreduce. job. queuename
mapred. job. reduce. input. buffer. percent	mapreduce. reduce. input. buffer. percent
mapred. job. reduce. markreset. buffer. percent	mapreduce. reduce. markreset. buffer. percent
mapred. job. reduce. memory. mb	mapreduce. reduce. memory. mb
mapred. job. reduce. total. mem. bytes	mapreduce. reduce. memory. totalbytes
mapred. job. reuse. jvm. num. tasks	mapreduce. job. jvm. numtasks
mapred. job. shuffle. input. buffer. percent	mapreduce. reduce. shuffle. input. buffer. percent
mapred. job. shuffle. merge. percent	mapreduce. reduce. shuffle. merge. percent
mapred. job. tracker. handler. count	mapreduce. jobtracker. handler. count
mapred. job. tracker. history. completed. location	mapreduce. jobtracker. jobhistory. completed. location
mapred. job. tracker. http. address	mapreduce. jobtracker. http. address
mapred. jobtracker. instrumentation	mapreduce. jobtracker. instrumentation
mapred. jobtracker. job. history. block. size	mapreduce. jobtracker. jobhistory. block. size
mapred. job. tracker. jobhistory. lru. cache. size	mapreduce. jobtracker. jobhistory. lru. cache. size
mapred. job. tracker	mapreduce. jobtracker. address
mapred. jobtracker. maxtasks. per. job	mapreduce. jobtracker. maxtasks. perjob
mapred. job. tracker. persist. jobstatus. active	mapreduce. jobtracker. persist. jobstatus. active
mapred. job. tracker. persist. jobstatus. dir	mapreduce. jobtracker. persist. jobstatus. dir
mapred. job. tracker. persist. jobstatus. hours	mapreduce. jobtracker. persist. jobstatus. hours
mapred. jobtracker. restart. recover	mapreduce. jobtracker. restart. recover
mapred. job. tracker. retiredjobs. cache. size	mapreduce. jobtracker. retiredjobs. cache. size
mapred. job. tracker. retire. jobs	mapreduce. jobtracker. retirejobs
mapred. jobtracker. taskalloc. capacitypad	mapreduce. jobtracker. taskscheduler. taskalloc. capacitypad
mapred. jobtracker. taskScheduler	mapreduce. jobtracker. taskscheduler
mapred. jobtracker. taskScheduler. maxRunning-TasksPerJob	mapreduce. jobtracker. taskscheduler. maxrunningtasks. perjob
mapred. join. expr	mapreduce. join. expr

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
mapred. join. keycomparator	mapreduce. join. keycomparator
mapred. lazy. output. format	mapreduce. output. lazyoutputformat. outputformat
mapred. line. input. format. linespermap	mapreduce. input. lineinputformat. linespermap
mapred. linerecordreader. maxlength	mapreduce. input. linerecordreader. line. maxlength
mapred. local. dir	mapreduce. cluster. local. dir
mapred. local. dir. minspacekill	mapreduce. tasktracker. local. dir. minspacekill
mapred. local. dir. minspacestart	mapreduce. tasktracker. local. dir. minspacestart
mapred. map. child. env	mapreduce. map. env
mapred. map. child. java. opts	mapreduce. map. java. opts
mapred. map. child. log. level	mapreduce. map. log. level
mapred. map. max. attempts	mapreduce. map. maxattempts
mapred. map. output. compression. codec	mapreduce. map. output. compress. codec
mapred. mapoutput. key. class	mapreduce. map. output. key. class
mapred. mapoutput. value. class	mapreduce. map. output. value. class
mapred. mapper. regex. group	mapreduce. mapper. regexmapper. . group
mapred. mapper. regex	mapreduce. mapper. regex
mapred. map. task. debug. script	mapreduce. map. debug. script
mapred. map. tasks	mapreduce. job. maps
mapred. map. tasks. speculative. execution	mapreduce. map. speculative
mapred. max. map. failures. percent	mapreduce. map. failures. maxpercent
mapred. max. reduce. failures. percent	mapreduce. reduce. failures. maxpercent
mapred. max. split. size	mapreduce. input. fileinputformat. split. maxsize
mapred. max. tracker. blacklists	mapreduce. jobtracker. tasktracker. maxblacklists
mapred. max. tracker. failures	mapreduce. job. maxtaskfailures. per. tracker
mapred. merge. recordsBeforeProgress	mapreduce. task. merge. progress. records
mapred. min. split. size	mapreduce. input. fileinputformat. split. minsize
mapred. min. split. size. per. node	mapreduce. input. fileinputformat. split. minsize. per. node
mapred. min. split. size. per. rack	mapreduce. input. fileinputformat. split. minsize. per. rack
mapred. output. compression. codec	mapreduce. output. fileoutputformat. compress. codec

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
mapred. output. compression. type	mapreduce. output. fileoutputformat. compress. type
mapred. output. compress	mapreduce. output. fileoutputformat. compress
mapred. output. dir	mapreduce. output. fileoutputformat. outputdir
mapred. output. key. class	mapreduce. job. output. key. class
mapred. output. key. comparator. class	mapreduce. job. output. key. comparator. class
mapred. output. value. class	mapreduce. job. output. value. class
mapred. output. value. groupfn. class	mapreduce. job. output. group. comparator. class
mapred. permissions. supergroup	mapreduce. cluster. permissions. supergroup
mapred. pipes. user. inputformat	mapreduce. pipes. inputformat
mapred. reduce. child. env	mapreduce. reduce. env
mapred. reduce. child. java. opts	mapreduce. reduce. java. opts
mapred. reduce. child. log. level	mapreduce. reduce. log. level
mapred. reduce. max. attempts	mapreduce. reduce. maxattempts
mapred. reduce. parallel. copies	mapreduce. reduce. shuffle. parallelcopies
mapred. reduce. slowstart. completed. maps	mapreduce. job. reduce. slowstart. completedmaps
mapred. reduce. task. debug. script	mapreduce. reduce. debug. script
mapred. reduce. tasks	mapreduce. job. reduces
mapred. reduce. tasks. speculative. execution	mapreduce. reduce. speculative
mapred. seqbinary. output. key. class	mapreduce. output. seqbinaryoutputformat. key. class
mapred. seqbinary. output. value. class	mapreduce. output. seqbinaryoutputformat. value. class
mapred. shuffle. connect. timeout	mapreduce. reduce. shuffle. connect. timeout
mapred. shuffle. read. timeout	mapreduce. reduce. shuffle. read. timeout
mapred. skip. attempts. to. start. skipping	mapreduce. task. skip. start. attempts
mapred. skip. map. auto. incr. proc. count	mapreduce. map. skip. proc-count. auto-incr
mapred. skip. map. max. skip. records	mapreduce. map. skip. maxrecords
mapred. skip. on	mapreduce. job. skiprecords
mapred. skip. out. dir	mapreduce. job. skip. outdir
mapred. skip. reduce. auto. incr. proc. count	mapreduce. reduce. skip. proc-count. auto-incr
mapred. skip. reduce. max. skip. groups	mapreduce. reduce. skip. maxgroups

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
mapred. speculative. execution. slowNode-Threshold	mapreduce. job. speculative. slownodethreshold
mapred. speculative. execution. slowTask-Threshold	mapreduce. job. speculative. slowtaskthreshold
mapred. speculative. execution. speculativeCap	mapreduce. job. speculative. speculativecap
mapred. submit. replication	mapreduce. client. submit. file. replication
mapred. system. dir	mapreduce. jobtracker. system. dir
mapred. task. cache. levels	mapreduce. jobtracker. taskcache. levels
mapred. task. id	mapreduce. task. attempt. id
mapred. task. is. map	mapreduce. task. ismap
mapred. task. partition	mapreduce. task. partition
mapred. task. profile	mapreduce. task. profile
mapred. task. profile. maps	mapreduce. task. profile. maps
mapred. task. profile. params	mapreduce. task. profile. params
mapred. task. profile. reduces	mapreduce. task. profile. reduces
mapred. task. timeout	mapreduce. task. timeout
mapred. tasktracker. dns. interface	mapreduce. tasktracker. dns. interface
mapred. tasktracker. dns. nameserver	mapreduce. tasktracker. dns. nameserver
mapred. tasktracker. events. batchsize	mapreduce. tasktracker. events. batchsize
mapred. tasktracker. expiry. interval	mapreduce. jobtracker. expire. trackers. interval
mapred. task. tracker. http. address	mapreduce. tasktracker. http. address
mapred. tasktracker. indexcache. mb	mapreduce. tasktracker. indexcache. mb
mapred. tasktracker. instrumentation	mapreduce. tasktracker. instrumentation
mapred. tasktracker. map. tasks. maximum	mapreduce. tasktracker. map. tasks. maximum
mapred. tasktracker. memorycalculatorplugin	mapreduce. tasktracker. resourcecalculatorplugin
mapred. tasktracker. reduce. tasks. maximum	mapreduce. tasktracker. reduce. tasks. maximum
mapred. task. tracker. report. address	mapreduce. tasktracker. report. address
mapred. task. tracker. task-controller	mapreduce. tasktracker. taskcontroller
mapred. tasktracker. taskmemorymanager. monitoring-interval	mapreduce. tasktracker. taskmemorymanager. monitoringinterval
mapred. tasktracker. tasks. sleeptime-before-sigkill	mapreduce. tasktracker. tasks. sleeptimebeforesigkill
mapred. temp. dir	mapreduce. cluster. temp. dir

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
mapred. text. key. comparator. options	mapreduce. partition. keycomparator. options
mapred. text. key. partitioner. options	mapreduce. partition. keypartitioner. options
mapred. textoutputformat. separator	mapreduce. output. textoutputformat. separator
mapred. tip. id	mapreduce. task. id
mapreduce. combine. class	mapreduce. job. combine. class
mapreduce. inputformat. class	mapreduce. job. inputformat. class
mapreduce. job. counters. limit	mapreduce. job. counters. max
mapreduce. jobtracker. permissions. supergroup	mapreduce. cluster. permissions. supergroup
mapreduce. map. class	mapreduce. job. map. class
mapreduce. outputformat. class	mapreduce. job. outputformat. class
mapreduce. partitioner. class	mapreduce. job. partitioner. class
mapreduce. reduce. class	mapreduce. job. reduce. class
mapred. used. genericoptionsparser	mapreduce. client. genericoptionsparser. used
mapred. userlog. limit. kb	mapreduce. task. userlog. limit. kb
mapred. userlog. retain. hours	mapreduce. job. userlog. retain. hours
mapred. working. dir	mapreduce. job. working. dir
mapred. work. output. dir	mapreduce. task. output. dir
min. num. spills. for. combine	mapreduce. map. combine. minspills
reduce. output. key. value. fields. spec	mapreduce. fieldsel. reduce. output. key. value. fields. spec
security. job. submission. protocol. acl	security. job. client. protocol. acl
security. task. umbilical. protocol. acl	security. job. task. protocol. acl
sequencefile. filter. class	mapreduce. input. sequencefileinputfilter. class
sequencefile. filter. frequency	mapreduce. input. sequencefileinputfilter. frequency
sequencefile. filter. regex	mapreduce. input. sequencefileinputfilter. regex
session. id	dfs. metrics. session-id
slave. host. name	dfs. datanode. hostname
slave. host. name	mapreduce. tasktracker. host. name
tasktracker. contention. tracking	mapreduce. tasktracker. contention. tracking
tasktracker. http. threads	mapreduce. tasktracker. http. threads
topology. node. switch. mapping. impl	net. topology. node. switch. mapping. impl

续表

Hadoop 1.0 属性名称	Hadoop 2.0 属性名称
topology.script.file.name	net.topology.script.file.name
topology.script.number.args	net.topology.script.number.args
user.name	mapreduce.job.user.name
webinterface.private.actions	mapreduce.jobtracker.webinterface.trusted

附表 3　HBase 默认配置说明

配置参数	说明
hbase.rootdir	设置 HRegionServer 的共享目录，默认值为：file:///tmp/hbase-${user.name}/hbase
hbase.master	HBase 的 HMaster 的端口，默认端口号为：60 000
hbase.cluster.distributed	设置 HBase 的运行模式，默认值为 false
hbase.tmp.dir	设置本地文件系统的临时文件夹，默认值为：/tmp/hbase-${user.name}
hbase.master.info.port	HMaster Web 界面端口，默认值为：16 010
hbase.master.info.bindAddress	HMaster Web 界面绑定的端口，默认值为：0.0.0.0
hbase.client.write.buffer	设置 HTable 客户端的写缓冲的默认大小，这个值越大，需要消耗的内存越大，默认：2 097 152
hbase.regionserver.port	HRegionServer 绑定的端口，默认：60 020
hbase.regionserver.info.port	HRegionServer Web 界面绑定的端口，默认：60 030
hbase.regionserver.info.port.auto	HMaster 或 HRegionServer 是否要动态搜一个可以用的端口来绑定界面，默认：false
hbase.regionserver.info.bindAddress	HRegionServer Web 界面的 IP 地址，默认：0.0.0.0
hbase.regionserver.class	HRegionServer 使用的接口，默认：org.apache.hadoop.hbase.ipc.HRegionInterface
hbase.client.pause	设置客户端暂停时间，默认：1000
hbase.client.retries.number	设置重试次数，默认为：10
hbase.client.scanner.caching	设置从服务端一次获取的行数，默认为：1
hbase.client.keyvalue.maxsize	一个 KeyValue 实例的最大 size，默认：10 485 760
hbase.regionserver.lease.period	客户端租用 HRegionServer 期限，即超时阀值，默认：60 000
hbase.regionserver.handler.count	RegionServers 受理的 RPC Server 实例数量，默认：10
hbase.regionserver.msginterval	HRegionServer 发消息给 HMaster 时间间隔，单位是 ms，默认：3000
hbase.regionserver.optionallogflushinterval	将 Hlog 同步到 HDFS 的间隔，默认：1000

续表

配置参数	说　　明
hbase. regionserver. regionSplitLimit	region 的数量到了这个值后就不会再分裂了，默认：2 147 483 647
hbase. regionserver. logroll. period	提交 commit log 的间隔，默认：3 600 000
hbase. regionserver. hlog. reader. impl	HLog file reader 的实现 默认：org. apache. hadoop. hbase. regionserver. wal. SequenceFileLogReader
hbase. regionserver. hlog. writer. impl	HLog file writer 的实现 默认：org. apache. hadoop. hbase. regionserver. wal. SequenceFileLogWriter
hbase. regionserver. thread. splitcompactcheckfrequency	HRegionServer 多久执行一次 split/compaction 检查，默认：20 000
hbase. regionserver. nbreservationblocks	储备的内存 block 的数量，默认：4
hbase. zookeeper. dns. interface	当使用 DNS 的时候，ZooKeeper 用来上报的 IP 地址的网络接口名字
hbase. zookeeper. dns. nameserver	当使用 DNS 的时候，ZooKeepr 使用的 DNS 的域名或者 IP 地址
hbase. regionserver. dns. interface	当使用 DNS 的时候，RegionServer 用来上报的 IP 地址的网络接口名字
hbase. regionserver. dns. nameserver	当使用 DNS 的时候，RegionServer 使用的 DNS 的域名或者 IP 地址
hbase. master. dns. interface	当使用 DNS 的时候，Master 用来上报的 IP 地址的网络接口名字
hbase. master. dns. nameserver	当使用 DNS 的时候，Master 使用的 DNS 的域名或者 IP 地址
hbase. balancer. period	HMaster 执行 region balancer 的间隔，默认：300 000
hbase. regions. slop	当任一 HRegionServer 有 average +（average × slop）个 region 时会执行 Rebalance
hbase. master. logcleaner. ttl	Hlog 存在于. oldlogdir 文件夹的最长时间，默认：600 000
hbase. master. logcleaner. plugins	LogsCleaner 服务器执行的一组 LogCleanerDelegat，值用逗号间隔的文本表示，默认：org. apache. hadoop. hbase. master. TimeToLiveLogCleaner
hbase. regionserver. global. memstore. upperLimit	单个 HRegionServer 的全部 memtores 的最大值，默认：0. 4
hbase. regionserver. global. memstore. lowerLimit	当强制执行 flush 操作的时候，当低于这个值的时候，flush 会停止，默认：0. 35
hbase. server. thread. wakefrequency	service 工作的 sleep 间隔，单位 ms，默认：10 000
hbase. hregion. memstore. flush. size	当 memstore 的大小超过这个值的时候，会 flush 到磁盘，默认：67 108 864

续表

配 置 参 数	说　　明
hbase. hregion. preclose. flush. size	当一个 region 中的 memstore 大于这个值的时候,我们又触发了 close. ,会先运行 pre-flush 操作,清理这个需要关闭的 memstore,然后将这个 region 下线,默认：5 242 880
hbase. hregion. memstore. block. multiplier	如果 memstore 有 hbase. hregion. memstore. block. multiplier 倍数的 hbase. hregion. flush. size 的大小,就会阻塞 update 操作,这是为了预防在 update 高峰期会导致的失控,默认：2
hbase. hregion. memstore. mslab. enabled	启用 memstore 分配本地缓冲区,默认：false
hbase. hregion. max. filesize	最大 HStoreFile 大小,默认：268 435 456
hbase. hstore. compactionThreshold	当一个 HStore 含有多于这个值的 HStoreFiles(每一个 memstore flush 产生一个 HStoreFile)的时候,会执行一个合并操作,把这 HStoreFiles 写成一个,默认：3
hbase. hstore. blockingStoreFiles	当一个 HStore 含有多于这个值的 HStoreFiles(每一个 memstore flush 产生一个 HStoreFile)的时候,会执行一个合并操作,update 会阻塞直到合并完成,直到超过了 hbase. hstore. blockingWaitTime 的值,默认：7
hbase. hstore. blockingWaitTime	hbase. hstore. blockingStoreFiles 所限制的 StoreFile 数量会导致 update 阻塞,这个时间是用来限制阻塞时间的,默认：90 000
hbase. hstore. compaction. max	每个"小"合并的 HStoreFiles 最大数量,默认：10
hbase. hregion. majorcompaction	一个 Region 中的所有 HStoreFile 的 major compactions 的时间间隔,默认是 1 天
hbase. mapreduce. hfileoutputformat. blocksize	MapReduce 中 HFileOutputFormat 可以写 storefiles/hfiles. 这个值是 hfile 的 blocksize 的最小值,默认：65 536
hfile. block. cache. size	分配给 HFile/StoreFile 的 block cache 占最大堆(-Xmx setting)的比例,默认：0. 2
hbase. hash. type	哈希函数使用的哈希算法,默认：murmur
hbase. master. keytab. file	HMaster server 验证登录使用的 kerberos keytab 文件路径
hbase. master. kerberos. principal	HMaster 运行需要使用 kerberos principal name
hbase. regionserver. keytab. file	HRegionServer 验证登录使用的 kerberos keytab 文件路径
hbase. regionserver. kerberos. principal	HRegionServer 运行需要使用 kerberos principal name
zookeeper. session. timeout	ZooKeeper 会话超时,HBase 把这个值传递改 zk 集群,向它推荐一个会话的最大超时时间,默认：180 000
zookeeper. znode. parent	ZooKeeper 中的 HBase 的根 ZNode,默认：/hbase
zookeeper. znode. rootserver	ZNode 保存的根 region 的路径,默认：root-region-server
hbase. zookeeper. quorum	ZooKeeper 集群的地址列表,用逗号分隔,默认：localhost
hbase. zookeeper. peerport	ZooKeeper 节点使用的端口,默认：2888



续表

配置参数	说　明
hbase. zookeeper. leaderport	ZooKeeper 用来选择 Leader 的端口，默认：3888
hbase. zookeeper. property. initLimit	ZooKeeper 的 zoo. conf 中的配置初始化 synchronization 阶段的 ticks 数量限制，默认：10
hbase. zookeeper. property. syncLimit	ZooKeeper 的 zoo. conf 中的配置发送一个请求到获得承认之间的 ticks 的数量限制，默认：5
hbase. zookeeper. property. dataDir	ZooKeeper 的 zoo. conf 中的配置快照的存储位置，默认：${hbase. tmp. dir}/zookeeper
hbase. zookeeper. property. clientPort	ZooKeeper 的 zoo. conf 中配置客户端连接端口，默认：2181
hbase. zookeeper. property. maxClientCnxns	ZooKeeper 的 zoo. conf 中的配置，默认：2000
hbase. rest. port	HBase REST Server 的端口，默认：8080
hbase. rest. readonly	定义 REST Server 的运行模式，false：所有的 HTTP 请求都是被允许的 - GET/PUT/POST/DELETE；true：只有 GET 请求是被允许的

附表 4　Hive 默认配置说明

配置参数	说　明
hive. exec. max. created. files	所有 Hive 运行的 Map 与 Reduce 任务可以产生的文件的和，默认值：100 000
hive. exec. dynamic. partition	是否为自动分区，默认值：false
hive. mapred. reduce. tasks. speculative. execution	是否打开推测执行，默认值：true
hive. input. format	Hive 默认的输入格式，默认值：org. apache. hadoop. hive. ql. io. CombineHiveInputFormat
hive. exec. counters. pull. interval	Hive 与 JobTracker 拉取 counter 信息的时间，默认值：1000ms
hive. script. recordreader	使用脚本时默认的读取类，默认值：org. apache. hadoop. hive. ql. exec. TextRecordReader
hive. script. recordwriter	使用脚本时默认的数据写入类，默认值：org. apache. hadoop. hive. ql. exec. TextRecordWriter
hive. mapjoin. check. memory. rows	内存里可以存储数据的行数，默认值：100 000
hive. mapjoin. smalltable. filesize	输入小表的文件大小的阈值，如果小于该值，就采用普通的 join，默认值：25 000 000
hive. auto. convert. join	是不是依据输入文件的大小，将 Join 转成普通的 Map Join，默认值：false
hive. mapjoin. followby. gby. localtask. max. memory. usage	Map Join 做 group by 操作时，可以使用多大的内存来存储数据，默认值：0. 55
hive. mapjoin. localtask. max. memory. usage	本地任务可以使用内存的百分比，默认值：0. 90

续表

配置参数	说　　明
hive. heartbeat. interval	在进行 MapJoin 与过滤操作时，发送心跳的时间，默认值：1000
hive. merge. size. per. task	合并后文件的大小，默认值：256 000 000
hive. mergejob. maponly	在只有 Map 任务的时候 合并输出结果，默认值：true
hive. merge. mapredfiles	正常 map-only job 后，是否启动 meroe job 来合并 map 端输出的结果，默认值：false
hive. hwi. listen. host	Hive UI 默认的 host，默认值：0. 0. 0. 0
hive. hwi. listen. port	UI 监听端口，默认值：9999
hive. exec. parallel. thread. number	Hive 可以并行处理 Job 的线程数，默认值：8
hive. exec. parallel	是否并行提交任务，默认值：false
hive. exec. compress. output	输出使用压缩，默认值：false
hive. mapred. mode	MapReduce 的操作的限制模式，操作的运行在该模式下没有什么限制，默认值：nonstrict
hive. join. cache. size	join 操作时，可以存在内存里的条数，默认值：25 000
hive. mapjoin. cache. numrows	Map Join 存在内存里的数据量，默认值：25 000
hive. join. emit. interval	有连接时 Hive 在输出前缓存的时间，默认值：1000
hive. optimize. groupby	在做分组统计时，是否使用 bucket table，默认值：true
hive. fileformat. check	是否检测文件输入格式，默认值：true
hive. metastore. client. connect. retry. delay	Client 连接失败时，retry 的时间间隔，默认值：1s
hive. metastore. client. socket. timeout	Client Socket 的超时时间，默认值：20s
mapred. reduce. tasks	每个任务 Reduce 的默认值，默认值：－1，表示自动根据作业的情况来设置 Reduce 的值
hive. exec. reducers. bytes. per. reducer	每个 Reduce 接受的数据量，默认值：1 000 000 000(1G)，如果送到 Reduce 的数据为 10G，那么将生成 10 个 Reduce 任务
hive. exec. reducers. max	Reduce 的最大个数，默认值：999
hive. metastore. warehouse. dir	默认的数据库存放位置，默认值：/user/hive/warehouse
hive. default. fileformat	默认的文件格式，默认值：TextFile
hive. map. aggr	Map 端聚合，相当于 Combiner，默认值：true
hive. exec. max. dynamic. partitions. pernode	每个任务节点可以产生的最大的分区数，默认值：100
hive. exec. max. dynamic. partitions	默认的可以创建的分区数，默认值：1000
hive. metastore. server. max. threads	Metastore 默认的最大的处理线程数，默认值：100 000
hive. metastore. server. min. threads	Metastore 默认的最小的处理线程数，默认值：200

续表

配置参数	说　　明
hive. exec. mode. local. auto	决定是否开启 Hive 的本地模式，默认值：true
hive. exec. mode. local. auto. inputbytes. max	如果 hive. exec. mode. local. auto 为 true，当输入文件小于此阈值时可以自动在本地模式运行，默认值：134 217 728L
hive. exec. mode. local. auto. tasks. max	如果 hive. exec. mode. local. auto 为 true，当 Hive Tasks (Hadoop Jobs)小于此阈值时，可以自动在本地模式运行，默认值：4
hive. auto. convert. join	是否根据输入小表的大小，自动将 Reduce 端的 Common Join 转化为 Map Join，从而加快大表关联小表的 Join 速度，默认值：false
hive. mapred. local. mem	Mapper/Reducer 在本地模式的最大内存量，以字节为单位，0 为不限制，默认值：0
mapred. reduce. tasks	所提交 Job 的 Reducer 的个数，使用 Hadoop Client 的配置，默认值：1
hive. exec. scratchdir	HDFS 路径，用于存储不同 Map/Reduce 阶段的执行计划和这些阶段的中间输出结果，默认值：/tmp/<user. name>/hive
hive. groupby. skewindata	决定 group by 操作是否支持倾斜的数据，默认值：false
hive. merge. mapfiles	是否开启合并 Map 端小文件，对于 Hadoop 0. 20 以前的版本，启用新的 Map/Reduce Job，对于 0. 20 以后的版本，则是使用 CombineInputFormat 的 MapOnly Job，默认值：true
hive. security. authorization. enabled	Hive 是否开启权限认证，默认值：false
hive. exec. plan	Hive 执行计划的路径，会在程序中自动进行设置，默认值：null
hive. exec. submitviachild	决定 Map/Reduce Job 是否应该使用各自独立的 JVM 进行提交(Child 进程)，默认情况下，使用与 HQL compiler 相同的 JVM 进行提交，默认值：false
hive. exec. script. maxerrsize	通过 TRANSFROM/MAP/REDUCE 所执行的用户脚本所允许的最大的序列化错误数，默认值：100 000
hive. exec. script. allow. partial. consumption	是否允许脚本只处理部分数据，如果设置为 true，因 broken pipe 等造成的数据未处理完成将视为正常，默认值：false
hive. exec. compress. intermediate	决定查询的中间 Map/Reduce job (中间 stage)的输出是否为压缩格式，默认值：false
hive. intermediate. compression. codec	中间 Map/Reduce Job 的压缩编解码器的类名(一个压缩编解码器可能包含多种压缩类型)，该值可能在程序中被自动设置
hive. intermediate. compression. type	设置中间 Map/Reduce job 的压缩类型，如"BLOCK"、"RECORD"
hive. exec. pre. hooks	语句层面，整条 HQL 语句在执行前的 hook 类名

续表

配 置 参 数	说　　明
hive. exec. post. hooks	语句层面，整条 HQL 语句在执行完成后的 hook 类名
hive. exec. dynamic. partition. mode	打开动态分区后，动态分区的模式，有 strict 和 nonstrict 两个值可选，strict 要求至少包含一个静态分区列，nonstrict 则无此要求，默认值：strict
hive. exec. default. partition. name	默认的动态分区的名称，当动态分区列为 null 时，使用此名称：'__HIVE_DEFAULT_PARTITION__'
hive. metastore. metadb. dir	Hive 元数据库所在路径
hive. metastore. uris	Hive 元数据的 URI，多个 thrift://地址，以英文逗号分隔
hive. metastore. connect. retries	连接到 Thrift 元数据服务的最大重试次数，默认值：3
hive. metastore. ds. connection. url. hook	JDO 连接 URL Hook 的类名，该 Hook 用于获得 JDO 元数据库的连接字符串，为实现了 JDOConnectionURLHook 接口的类
javax. jdo. option. ConnectionPassword	JDO 的连接密码
javax. jdo. option. ConnectionURL	元数据库的连接 URL
hive. metastore. ds. retry. attempts	当没有 JDO 数据连接错误后，尝试连接后台数据存储的最大次数，默认值：1
hive. metastore. ds. retry. interval	每次尝试连接后台数据存储的时间间隔，以 ms 为单位，默认值：1000

参考文献

[1] Nature. Big Data. http://www.nature.com/news/specials/bigdata/index.html.

[2] Bryant R, Katz R H, Lazowska E D. Big-data computing: Creating Revolutionary Breakthroughs in Commerce, Science, and Society. In Computing Research Initiatives for the 21st Century, Computing Research Association, Ver. 8, 2008. http://www.cra.org/ccc/docs/init/Big_Data.pdf.

[3] Science. Dealing with Data. http://www.sciencemag.org/site/special/data/.

[4] James Manyika, Michael Chui, Brad Brown. Big data: The next frontier for innovation, competition, and productivity. http://www.mckinsey.com/Insights/MGI/Research/Technology_and_Innovation/Big_data_The_next_frontier_for_innovation.

[5] Richard L Villars, Carl W Olofson, Matthew Eastwood. Big Data: What It Is and Why You Should Care. Sponsored by: June 2011. http://sites.amd.com/us/Documents/IDC_AMD_Big_Data_Whitepaper.pdf.

[6] World Economic Forum. Big Data, Big Impact: New Possibilities for International Development. http://www3.weforum.org/docs/ WEF_TC_MFS_BigDataBigImpact_Briefing_2012.pdf.

[7] D Agrawal, P Bernstein, E Bertino, et al. Challenges and Opportunities with Big Data. A community white paper. http://cra.org/ccc/docs/init/bigdatawhitepaper.pdf.

[8] The White House. the Obama Administration is announcing the Big Data Research and Development Initiative. http://www.whitehouse.gov/sites/default/files/microsites/ostp/big_data_press_release_final_2.pdf.

[9] Global Pulse White Paper. Big Data for Development: Opportunities & Challenges. http://www.unglobalpulse.org.

[10] Big data. http://en.wikipedia.org/wiki/Big_data.

[11] IDC's Worldwide Big Data Taxonomy. 2011 published by IDC in October 27, 2011.

[12] IBM. What is big data? [EB/OL]. [2012-10-02], http://www-01.ibm.com/software/data/bigdata/.

[13] Big data. http://www.gartner.com/it-glossary/big-data/.

[14] Big Data Definitions. v1, Developed at Jan. 15-17, 2013 NIST Cloud/BigData Workshop.

[15] John F Gantz, David Reinsel, Christopher Chute, et al. IDC - the expanding digital universe: A forecast of worldwide information growth through 2010. Technical report, March 2007.

[16] Gartner Business Intelligence. The Grand Challenges of Information - Innovating to Make Your Infrastructure and Users Smarter. January 31 - February 1, 2011.

[17] David Lazer, Ryan Kennedy, Gary King. et al. The Parable of Google Flu: Traps in Big Data Analysis. Science, 14 March 2014: Vol. 343 no. 6176 pp. 1203-1205 DOI: 10.1126/science.1248506.

[18] Ginsberg J, Mohebbi MH, Patel RS, et al. Detecting influenza epidemics using search engine query data. Nature 457(7232): 1012-4. 2009.

[19] Charles Duhigg. How Companies Learn Your Secrets. 2. 2012

[20] 艾瑞咨询集团. 大数据行业应用展望报告. 28/10/2013.

[21] Flume 主页：https://github.com/cloudera/flume/.
[22] scribe 主页：https://github.com/facebook/scribe.
[23] kafka 主页：http://kafka.apache.org/.
[24] 淘宝 Time Tunnel 主页：http://code.taobao.org/p/TimeTunnel/src/.
[25] chukwa 主页：http://incubator.apache.org/chukwa/.
[26] Scribe 日志收集系统介绍：http://dongxicheng.org/search-engine/scribe-intro/.
[27] J Kreps，N Narkhede，and J Rao. Kafka：A distributed messaging system for log processing. In Proceedings of 6th International Workshop on Networking Meets Databases (NetDB)，Athens，Greece，2011.
[28] Feiyi Wang，Sarp Oral，Galen Shipman，et al. Understanding Lustre Filesystem Internals. Technical Report ORNL/TM-2009/117.
[29] Lustre File System[EB /OL]. http://lustre.opensfs.org/.
[30] Lustre 2.0 Operations Manual. http://wiki.lustre.org/manual/LustreManual20_HTML/index.html.
[31] Sanjay Ghemawat，Howard Gobioff，and Shun-Tak Leung. The Google File System. SOSP，Oct 2003.
[32] PVFS：Parallel Virtual File System. http://www.pvfs.org/documentation/.
[33] Sage A Weil，Scott A Brandt，Ethan L Miller，et al. In Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06).
[34] Hadoop Distributed File System. http://wiki.apache.org/hadoop/HDFS.
[35] Hadoop Distributed File System：Architecture & Desig. http://hadoop.apache.org/docs/r0.18.0/hdfs_design.pdf.
[36] Chang F，Dean J，Ghemawat S，et al. Bigtable：A distributed storage system for structured data. In：Proc. of the OSDI. New York：ACM Press，2006.
[37] Bigtable-like structured storage for hadoop HDFS. http://wiki.apache.org/hadoop/Hbase.
[38] De Candia G，Hastorun D，Jampani M，et al. Dynamo：Amazon's highly available key-value store. In：Proc. of the SOSP. New York：ACM Press，2001：205-220.
[39] Lakshman A，Malik P. Cassandra-A decentralized structured storage system. 2009. http://www.cs.cornell.edu/projects/ladis2009/ papers/lakshman-ladis2009.pdf.
[40] Tokyo Cabinet. http://fallabs.com/tokyocabinet/.
[41] Apache CouchDB：The apache CouchDB project. http://couchdb.apache.org/.
[42] MongoDB. http://www.mongodb.org.
[43] Redis. http://redis.io/.
[44] Brewer. Towards Robust Distributed Systems. (Invited Talk)，in Principles of Distributed Computing，Portland，Oregon，2000.
[45] Gilbert S，Lynch N. 2002. Brewers conjecture and the feasibility of consistent，available，partition-tolerant Web services. ACM SIGACT News 33(2).
[46] http://en.wikipedia.org/wiki/ACID.
[47] Architecture White Paper，Greenplum Database：Critical Mass Innovation. http://www.greenplum.com.
[48] Pritchett D. BASE：An acid alternative. 2008.
[49] Srini Penchikala. NoSQL Database Adoption Trends. http://www.infoq.com/.

[50] Min Chen, S Mao, Y Liu. Big Data: A Survey. ACM/Springer Mobile Networks and Applications (ACM MONET), Vol. 19, No. 2, pp. 171-209, April 2014.

[51] Andrew NG. Logistic Regression Classification.

[52] 冯登国,张敏,李昊. 大数据安全与隐私保护. 计算机学报,2014,01:246-258.

[53] Cloud Security Alliance Big Data Working Group. Expanded Top Ten Big Data Security and Privacy Challenges. https://cloudsecurityalliance.org/research/big-data/#_downloads.

[54] Burton Bloom. Space/Time trade-offs in Hash Coding with Allowable Errors. Communications of the ACM, 1970.

[55] Hinton G E, Osindero S and Teh Y. A fast learning algorithm for deep belief netsNeural Computation. 2006,18:1527-1554.

[56] Yoshua Bengio, Pascal Lamblin, Dan Popovici, et al. Greedy Layer-Wise Training of Deep Networks. in J. Platt et al. (Eds), Advances in Neural Information Processing Systems 19 (NIPS 2006), pp. 153-160, MIT Press, 2007.

[57] Marc Aurelio Ranzato, Christopher Poultney, Sumit Chopra, et al. Efficient Learning of Sparse Representations with an Energy-Based Model, in J. Platt et al. (Eds), Advances in Neural Information Processing Systems (NIPS 2006), MIT Press, 2007.

[58] Cheng Xue-Qi, Jin Xiao-Long, Wang Yuan-Zhuo, et al. Survey on Big Data System and Analytic Technology. Ruan Jian Xue Bao/Journal of Software, 2014, 25(9).

[59] Sun Da-Wei, Zhang Guang-Yan, Zheng Wei-Min. Big Data Stream Computing: Technologies and Instances. Ruan Jian Xue Bao/Journal of Software, 2014, (4): 839-862.

[60] Dean Jeff and Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. 2004.

[61] M Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. Proc. 7th Symp. Operating Systems Design and Implementation (OSDI 06), Usenix, 2006.

[62] Pike Rob (Google Inc., CA, United States), Dorward Sean, Griesemer Robert, et al. Interpreting the data: Parallel analysis with Sawzall. Scientific Programming, v13, n4, SPEC. ISS, p277-298, 2005.

[63] http://hadoop.apache.org/docs/stable/index.html.

[64] 董西成:Hadoop 技术内幕:深入解析 YARN 架构设计与实现原理. 北京:机械工业出版社,2013.

[65] Owen O'Malley, Kan Zhang, Sanjay Radia, et al. Hadoop Security Design. 2009.

[66] Dean, Jeffrey & Ghemawat, Sanjay. MapReduce: Simplified Data Processing on Large Clusters. Retrieved Apr. 6, 2005.

[67] Vavilapalli, Vinod Kumar, Murthy, et al. Apache hadoop YARN: Yet another resource negotiator. Proceedings of the 4th Annual Symposium on Cloud Computing, SoCC 2013.

[68] ResourceManager High-Availability. https://issues.apache.org/jira/browse/YARN-149.

[69] Apache Hadoop YARN-ResourceManager. http://hortonworks.com/blog/apache-hadoop-yarn-resourcemanager/.

[70] Apache Hadoop YARN-NodeManager. http://hortonworks.com/blog/apache-hadoop-yarn-nodemanager/.

[71] Hadoop MapReduce Next Generation - Writing YARN Applications. http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/WritingYarnApplications.html.

[72] http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[73] M Zaharia, D Borthakur, J S Sarma, et al. Job scheduling for multi-user mapreduce clusters. EECS Department, University of California, Berkeley, Tech. Rep. , Apr 2009.

[74] http://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html.

[75] J Polo, D Carrera, Y Becerra, et al. Performance-driven task co-scheduling for mapreduce environments. Network Operations and Management Symposium (NOMS), 2010 IEEE, 2010: 373-380.

[76] K Kc and K Anyanwu. Scheduling hadoop jobs to meet deadlines. 2nd IEEE International Conference on Cloud Computing Technology and Science (CloudCom), 2010: 388-392.

[77] A Ghodsi, M Zaharia, B Hindman, et al. Dominant Resource Fairness: Fair Allocation of Multiple Resources Types. NSDI 2011, March 2011.

[78] L Neumeyer, B Robbins, A Nair, et al. S4: Distributed stream computing platform. Data Mining Workshops, International Conference on, 0:170-177,2010.

[79] Malewicz G, Austern M H, Bik A J, et al. Pregel: a system for large-scale graph processing. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing (PODC'09), pp. 6-146 (2009).

[80] The Apache HBase™ Reference Guide. http://hbase.apache.org/.

[81] Lars George. HBase: The Definitive Guide. O'Reilly Media, Inc, USA,2011.

[82] Enis Soztutar. Apache HBase Region Splitting and Merging. http://zh.hortonworks.com/blog/apache-hbase-region-splitting-and-merging/, February 1st, 2013.

[83] Apache Hive™. http://hive.apache.org/.

[84] Hive Data Definition Language. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL.

[85] Hive Data Manipulation Language. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DML.

[86] Chu CT, Kim SK, Lin YA, et al. Map-Reduce for machine learning on multicore. In: Scholkopf B, Platt JC, Hoffman T, eds. Proc. of the NIPS. Vancouver: MIT Press, 2006: 281,288.

[87] Haoyuan Li, et al. Parallel FP-Growth for Query Recommendation. RecSys '08 Proceedings of the 2008 ACM conference on Recommender systems. 2008.

[88] http://www.ibm.com/developerworks/cn/web/1103_zhaoct_recommstudy2/index.html.

[89] https://www.ibm.com/developerworks/cn/java/java-lo-mapreduce/.